The brick wall: *NP*-completeness

Coping with problems computers can't solve

John Franco

For over 40 years, artificial intelligence (AI) has not fullfilled its promise of truly intelligent machines for general use. As early as the 1950s and 1960s, scientists developed computational models of intelligence. They then excitedly coded these models into the best computers of the day. At first the scientists were puzzled by the machines' inability to produce reasoned output. Bewilderment became frustration when they realized they had banged into an unforeseen brick wall. This wall had stopped them in their tracks and continues to do so today.

AI also has stymied scientists and engineers in other fields such as operations research (the field concerned with determining efficient manufacturing and scheduling protocols), VLSI chip design and testing, and data base management. The brick wall exists because many combinatorial problems that are fundamentally important are *NP*-complete. (*NP* stands for Nondeterministic Polynomial time.)

The thick brick wall

We illustrate the problem *NP*-completeness causes with an example taken from manufacturing. A general purpose welding device is to be used on an assembly line. It will make numerous welds at predetermined positions on a particular kind of part. The positions will be welded in a specific sequence called a *schedule.* This schedule is programmed into the welder before a "run."

Possibly as many as 1000 welds can be scheduled for each part that passes on the assembly line. Clearly the speed, and thus the cost, of producing a product depends on how fast the welder can finish all its welds on each part and return to its starting point. But the welder's speed depends on the distances traversed from one position to the next. What is needed is an optimal schedule for the positioning of the weld-head: that is, a schedule that minimizes the total distance traversed by the weld-head to finish each part.

Let's consider how we might find such an optimal schedule. A simple algorithm is the following: tabulate all possible schedules along with the total distance traversed and search for the schedule offering the least total distance. This seems like a perfectly reasonable approach. But when we try to implement it we run into the thick brick wall.

The number of schedules that we have to tabulate can easily be seen to be the factorial of the number of positions needing welds. (From now on $n!$ will denote this number where n is the number of positions needing welds.) If we need even as few as 30 welds, $n!$ is 30! which is greater than 10^{30}. On a supercomputer that can tabulate schedules at the rate of one per 10^{-12} seconds (which is very, very fast!!), we can tabulate our task involving 30 welds in no less than $10^{30} \times 10^{-12}$ seconds. But this is about 30 years.

Jackhammers don't work

Before we search for a better algorithm, we must understand what is holding us back. The problem we have with tabulating possible solutions is that, if n is increased by 1, the size of the table, and therefore the amount of time needed to find an optimal schedule, more than doubles. This phenomenon is known as *the exponential explosion of complexity.* (Here, the word complexity refers to the running time of an algorithm.) An algorithm suffering from this phenomenon has a complexity at least 2^n.

What we need is an algorithm with a complexity n^2 or, better yet, n. If we had a scheduling algorithm of complexity n^2, we could solve our scheduling problem for 1000 welds in less

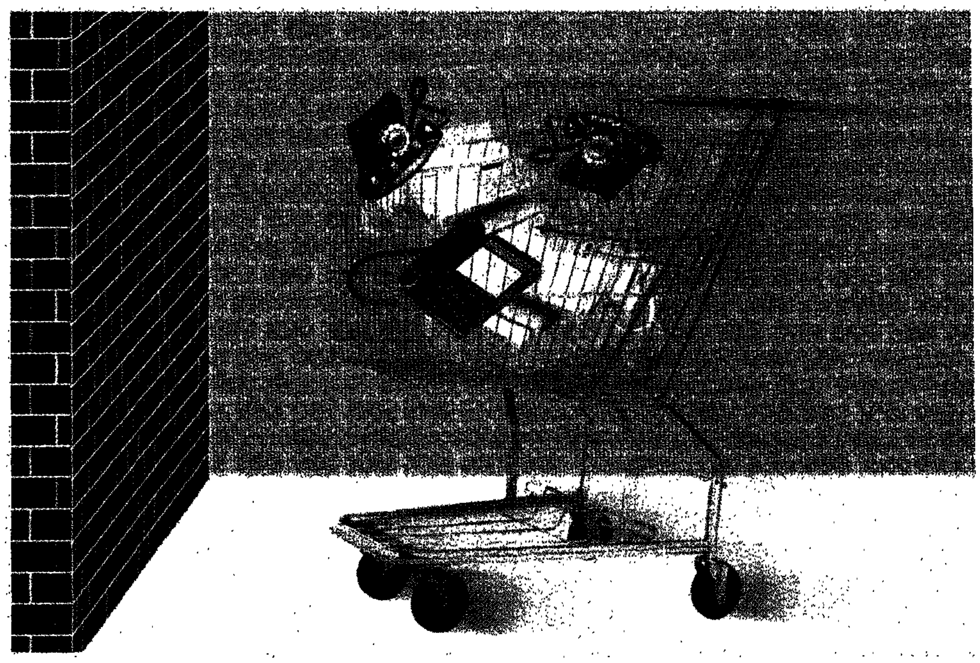


Illustration by Debbie McIntyre based on © 1996 PhotoDisc, Inc. stock art.

0278-6648/97/$10.00 © 1997 IEEE

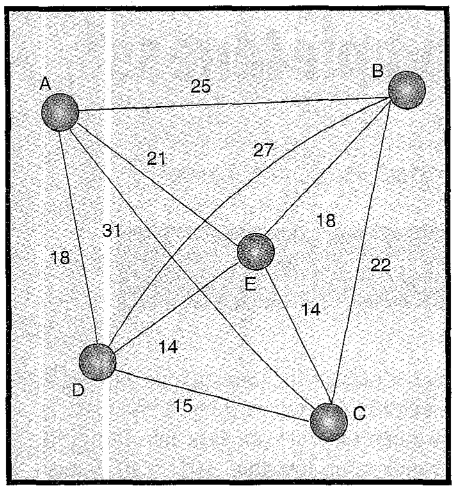


Fig. 1 Finding an optimal weld schedule.

than 1 second with a Pentium processor. With a complexity of n we could get a result in less than 1 second on an IBM XT.

How can we find an algorithm of complexity n^2 or n? We can achieve this complexity if there is a way to iteratively extend a partial solution in such a way that the extensions do not deviate from the optimal solution.

Let's try the obvious algorithm, called the greedy method (because it always chooses a next weld point based on some minimum weld value), for extending a weld schedule: start with a partial schedule of one weld position chosen arbitrarily; repeatedly append to the partial schedule the weld position that is closest to the last position in the partial schedule and that has not yet been added to the partial schedule; when all positions are in the partial schedule, return the partial schedule as a full schedule. The complexity of this algorithm is n^2 since the distance between all pairs of positions must be considered. The question is whether this algorithm always produces an optimal schedule.

We investigate this question by considering the simple example of Fig. 1, which represents five weld positions denoted A, B, C, D, and E. Lines and numbers are used to show distances: a line designates a pair of positions, and the number associated with that line gives the distance between that pair. (Note: the numbers given cannot represent Euclidean distances if the five points are in the same plane. It is typically the case that the time needed to move an object is not proportional to Euclidean distance because it depends on other factors such as direction of movement.)

We will apply the greedy method to Fig. 1 starting at point A. We create a partial schedule containing A, then append D to the schedule since the distance to D from A is 18. Also, this is less than the distance from A to any other unscheduled point.

Next we append E because the lowest distance from D to an unscheduled point is 14 (to E). Next C and finally B are scheduled and the algorithm returns the schedule A-D-E-C-B. The total distance covered by this schedule is 18+14+14+22+25=93. However, a better schedule would be A-D-C-E-B which covers a distance of 18+15+14+18+25=90. This shows that our greedy method does not necessarily produce an optimal schedule.

Can some other n^2 complexity algorithm for obtaining optimal schedules be found? For this problem, no n^2, or n^3, or n^{10} or even n^{100} algorithm has been found despite decades of searching. In fact, many believe no n^k algorithm, where k is a constant, will ever be found for this problem. (An algorithm of complexity n^k, where k is a constant, is said to be efficient.)

This pessimism extends to all *NP*-complete problems. Unfortunately, most interesting real-world problems are *NP*-complete. Therefore, this is a very serious situation and something must be done about it.

Going around the wall

The astute reader may be wondering whether parallel processors are the answer. Perhaps we can achieve real-time solutions to *NP*-complete problems by dividing the computation required over many, many processors.

Parallelization can help but cannot solve the problem altogether. The reason is there are not enough atoms in the universe. Assume that every processor requires at least one atom and that there are fewer than 2^{200} atoms in the universe. (This is just a guess.) A weld scheduling algorithm of complexity 2^n distributed evenly over 2^{200} processors (the maximum possible) operating at supercomputer speed would compute for more than one century if n is only 235. Clearly, another approach is needed.

Another possibility is to make machines faster. Unfortunately, we seem to run into a barrier here, too. For several reasons, it is unlikely that computers, as we know them, will ever be able to perform even an elementary operation in less than 10^{-1} seconds. (This is just 1000 times faster than the time we assumed for the supercomputer.)

A major reason has to do with the impedance of the interconnections between semiconductor devices. But, even if this problem is solved, the following simple argument shows what we are up against. The time it takes for a signal to cross a silicon-based transistor is given roughly by $t_D = d/s_0$ where s_0 is 10^{11}µm/sec, and d is the distance the signal must travel. With current technology, distances of.18µm are just now being reached successfully. This translates to about 2×10^{-12} seconds.

To reach 10^{-15} seconds, features would have to be 2000 times smaller than current technology allows. But, that would require features to be much less than 1 atom thick. If speeds are limited to 10^{-15} seconds, the optimal weld scheduling algorithm given earlier could just handle almost 20 more weld positions than it can with current computers.

These arguments show that improved hardware will probably do little to help us cope with *NP*-completeness. However, there is a software solution that often works. Use an algorithm of n^2 or n complexity to find an answer that *approximates* the optimal solution.

We saw with Fig. 1 that the greedy method returned a solution totaling 93 while the optimal solution's distance value was 90. Thus, the greedy method's solution was only slightly more than 3% away from the optimal value. If we can guarantee a solution that is less than 4% away from the value of the optimal solution, then the greedy method might have tremendous practical use.

Unfortunately, the best we can guarantee for optimal schedules, assuming distances are Euclidean, is to be within 50% of the optimal value; this is, by the way, accomplished by an algorithm other than the greedy method. In the general case, when distances are not necessarily Euclidean, efficiently finding an approximation that is guaranteed to be within a fixed percentage of the optimal is impossible. (Unless, there exists an efficient algorithm for finding the optimal solution.)

Fortunately, good approximation algorithms exist for many *NP*-complete problems. For example, consider the bin packing problem (described in the following example). The telephone company needs to store telephones of various sizes in a warehouse. The warehouse contains a number of bins of identical capacity. The question is how to organize the telephones in the bins so as to use the minimum number of bins.

An efficient approximation algorithm for this problem is known as the *First-Fit Decreasing* (FFD) heuristic. The idea is this: number bins 1,2,3,... arbitrarily; repeatedly put the largest sized, unstored telephone into the lowest number bin with the remaining capacity greater than the size of the telephone. (That is, the phone is placed in the first bin in which it fits.) It can be shown that FFD will never cause more than 22% extra bins to be used. Figure 2 shows that we can be off by as much as 22% when the number requiring storage is large.

Before continuing, let's answer a question that frequently pops up at this time: how can you prove an approximation algorithm guarantees a certain performance if you don't know what the optimal solution is? Or, asking the question another way: if you can prove a performance guarantee, can't you then find the optimal solution?

The answer is no. We illustrate using the *First-Fit* (FF) heuristic, which is FFD. However, it does not force consideration of largest sized telephones first. For FF, no more than one bin can be less than half full. If this were not the case, we could identify the first two bins, i and j, with $i < j$, that are less than half full; but, then FF would have put the telephones of bin j into bin i (or possibly some other bins).

On the other hand, an optimal solution just completely fills all the bins needed for a solution. Therefore, FF produces a solution requiring no more than twice the number of bins taken by the optimal solution plus 1. Not all performance arguments are this simple; some require more than 50 pages. But this argument shows you can compute performance guarantees without knowing the optimal solution.

Probabilistic and empirical results

Sometimes, the performance guarantees of approximation algorithms are much worse than is actually encountered in practice. Often, this is because locally optimal components can be constructed and put together to produce optimal or near-optimal solutions.

The bin packing problem provides a great example of this. Suppose telephone sizes are uniformly distributed from size 0 to the bin's capacity. That means any one size is as likely to exist as any other from size 0 up to the bin's capacity. (This is not reasonable, but the idea can be generalized to support similar results under more reasonable distributions.) Then, for any size greater than the mean, there is—with high probability—a size below the mean such that the sum of the two sizes is either exactly the bin's capacity or extremely close. Each such pairing can fill a bin nearly or exactly to capacity. Probabilistically, we can find such pairings filling nearly all bins to capacity or just below capacity. We can prove that under a uniform distribution only a constant number of bins are wasted, with probability tending to 1, using FFD. The same results hold for a very wide variety of distributions. Therefore, bin packing is regarded to be an easily solved problem.

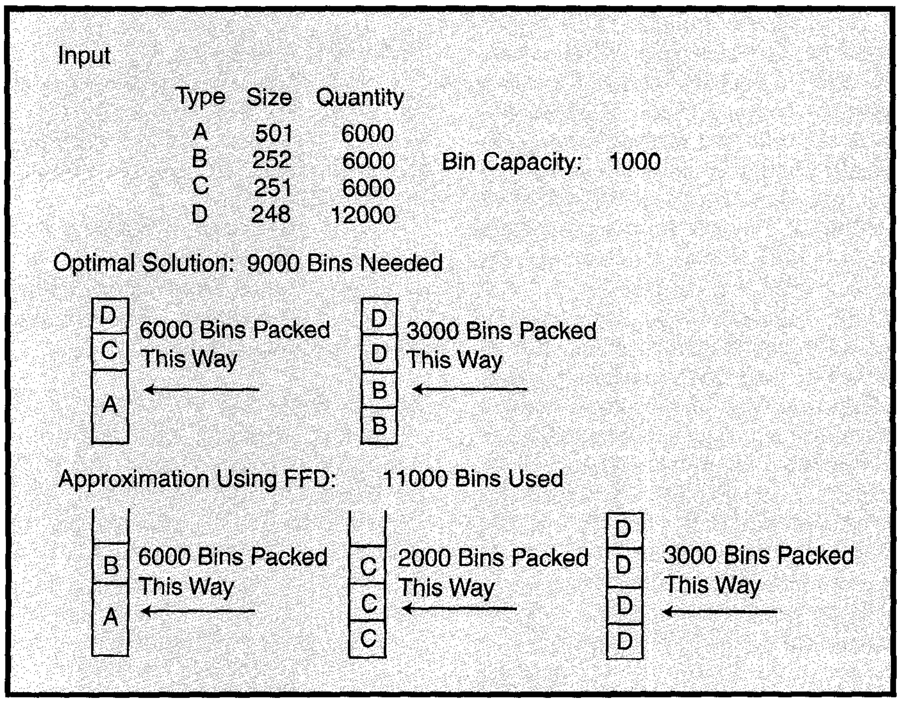
However, many *NP*-complete problems do not present an obvious way to combine locally optimal components. Algorithms for these may show good performance sometimes and poor performance other times.

To illustrate, we return to the field of artificial intelligence and consider the Satisfiability (SAT) problem. An instance of SAT is a Boolean formula in Conjunctive Normal Form (CNF); that is, an "AND" of expressions which are "OR"s of Boolean variables and their complements. Figure 3 gives a sample CNF formula.

In this example, the Boolean variables are x_1, x_2, x_3, x_4, x_5, and x_6. Each can be assigned only one of two values: *true* or *false*. If variable x is assigned value *true* (*false*), then its complement x has value *false* (*true*). For any variable x, both x and x are called *literals* where (and if) they appear in a formula.

Expressions such as $(x_1 \vee x \vee x_3)$ are called clauses. A clause has value *true* if and only if one of its literals has value *true*. A formula has value *true*, or is satisfiable, if and only if all its clauses have value *true* for some assignment of values to its variables.

The SAT problem is to determine whether or not a given formula is satisfiable. The assignment $x_1 = x_2 = x_4 = x_5 = x_6 =$ *true*, and $x_3 =$ *false* causes all clauses of the formula in Fig. 3 to have value *true*. On the other hand, any assignment with $x_1 = x_3 =$ *false*, and $x_2 =$ *true* causes the first clause to have value *false*.



Input

Type	Size	Quantity
A	501	6000
B	252	6000
C	251	6000
D	248	12000

Bin Capacity: 1000

Optimal Solution: 9000 Bins Needed

D
C
A
6000 Bins Packed This Way
D
D
B
B
3000 Bins Packed This Way

Approximation Using FFD: 11000 Bins Used

B
A
6000 Bins Packed This Way
C
C
C
2000 Bins Packed This Way
D
D
D
D
3000 Bins Packed This Way

Fig. 2 A bin packing example showing optimal solution and FFD approximation.

SAT is probably the most well-known *NP*-complete problem of all. SAT was the first problem shown to be *NP*-complete and has been used to show others are *NP*-complete. In real life, it shows up in VLSI testing and design, in theorem proving, in preventing catastrophic machine shutdowns in case of part failure. SAT also shows up in many other aspects of science, engineering, operations research and artificial intelligence.

There is no known way to combine locally optimum components to solve a general SAT problem; there seems to be a deep interdependence between large subsets of clauses, especially when formulas are not satisfiable. SAT is further handicapped by the fact that the concept of approximation algorithm is meaningless: either an input formula is satisfiable or not. We must determine precisely which it is or we have made a big mistake. Therefore, SAT is not regarded as an easy problem in general, and current research aims to improve this situation.

One research direction attempts to uncover special classes of SAT that can be solved provably by an efficient algorithm. Remarkably, there are many such classes. Here we mention a few.

If a formula is restricted so that every clause contains at most two literals, it can be solved in time proportional to the size of the formula by a special algorithm. Such formulas are called 2-SAT formulas. Interestingly, formulas containing at most three literals per clause (3-SAT) are *NP*-complete. No one really understands why adding one more literal to each 2-SAT clause has such an effect on complexity. Adding a literal to clauses containing one literal changes nothing in this regard.

If a formula is restricted so that at most one literal in every clause is not negated, then the formula can be solved in time proportional to the size of the formula. Such formulas are called *Horn* formulas. Several efficiently solved classes are extensions of Horn formulas. There are also hierarchies of formulas such that a formula at level k in the hierarchy can be solved in time 2^k times the square of the length of the formula.

If there is an efficient way to check whether a formula belongs to such a special class of SAT, this check can be applied before using a general purpose SAT solver. In case the check succeeds, a special purpose and efficient algorithm can be brought to bear.

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_6) \wedge (\bar{x}_3 \vee x_5 \vee x_6) \wedge (x_3 \vee x_4 \vee x_6) \wedge (\bar{x}_4 \vee \bar{x}_5 \vee \bar{x}_6)$$

Fig. 3 A boolean formula in conjunctive normal form

It seems that identifying efficiently solved special classes of SAT helps some but not much. In particular, this approach usually fails when input formulas are not satisfiable. This is illustrated by probabilistic and empirical results obtained on random 3-SAT formulas with m clauses and n variables. Random 3-SAT formulas contain m clauses selected uniformly and independently from the set of all possible clauses that can be constructed from n variables. There are such possible clauses.

Experiments have shown a random 3-SAT formula is satisfiable, in probability, if m and n are set so that $m/n < 4.2$, and is not satisfiable, with probability tending to 1 with increasing m, n, if $m/n > 4.3$. We know, through probabilistic analysis, that a searching algorithm finds a satisfying assignment efficiently, with probability tending to 1, when $m/n < 3.003$. But, so far, a random 3-SAT formula is known to be a member of some efficiently solved special class of SAT, with high probability, only if $m/n < 1$.

No one has been able to find a good algorithm (in some probabilistic sense) for verifying unsatisfiability for a vast range of formula types. Specifically, there is no known algorithm that efficiently verifies the unsatisfiability of a random formula, with probability tending to 1, when m/n is a constant greater than 4.3 or even when m/n grows as fast as n^e, $e < 1$.

This seems to be a consequence of the following statement which can be proven by induction: a minimally unsatisfiable set of p clauses must contain fewer than p distinct variables. (Negated or unnegated occurrences are treated the same.) A set of clauses is minimally unsatisfiable if it is unsatisfiable and removal of any clause from the set leaves a satisfiable set of clauses.

Even *resolution* fails to do a good job in this range. Resolution is a process that iteratively creates new clauses that do not change the satisfiability of a formula if added to it. If the empty clause is created during resolution, the formula is unsatisfiable. Resolution of some form is commonly used in SAT algorithms.

One achievement of probabilistic analysis is that it shows exactly why resolution fails: this has to do with the "sparsity" of random 3-SAT formulas. However, probabilistic analysis has yet to help find a method that succeeds. Research on this question is ongoing and involves people from all over the world. If a fast method, in some probabilistic sense, is found, it may greatly impact artificial intelligence and other fields. But with all the thought already given to this, a successful algorithm will probably be unlike any other.

Acknowledgments

Supported in part by the Office of Naval Research, Nooo14-94-1-0382.

Read more about it

• A very good discussion of the theory of *NP*-completeness and approximation algorithms is Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco (1979), ISBN:0-7167-1045-5.

• For a discussion of the physical limits of devices see Robert W. Keyes, "Power dissipation in information processing," in *Science*, Volume 168 (May 15,1970).

• For a discussion of Satisfiability algorithms see Jun Gu, Paul W. Purdom, John Franco, Ben Wah, *Satisfiability Algorithms*, Cambridge University Press (1998).

About the author

Dr. John Franco in 1981 received a Ph.D. in Computer Science from Rutgers University (NJ). He was a member of the Digital Signal Processing group at Bell Telephone Laboratories from 1969 to 1976. There he invented a multiple FSK digital demodulator. He has served at the University of Cincinnati (OH) since 1990. He is best known for his seminal work on the probabilistic analysis of algorithms for the Satisfiability problem.

Global implications

A survey taken in the spring of 1996 by NACE (National Association of Colleges and Employers) showed that computer science majors were more likely to have chosen their field for its earning potential; more than one-third (37%) of these students who responded said that money was the deciding factor. Their average starting salary with a bachelor's degree then was $34,565 (US).

Well, things only got better for the Class of '97. NACE reports in its recent September *Salary Survey* that computer science undergrads have seen their average offer rise to $37,215 (US). They were sought after by employers of all types and sizes.

Computer engineering undergrads also continue to do exceptionally well. Their average offer jumped 6.8 percent to $40,093 (US). Computer and business equipment manufacturers accounted for 22 percent of the offers; but computer software/data processing employers (part of the service sector) were next with 18 percent of the offers.

However, electrical engineers shouldn't feel bad. BSEEs saw a 4 percent increase in their average offer, which now stands at $39,546 (US). Half their offers came from: electrical/electronics manufacturers (27%); computer and business equipment manufacturers (13%); and aerospace companies (10%). Another third came from the service sector, primarily those related to computers and technology, e.g., communication service companies, computer software/data processing firms and consulting services organizations.

These good times are far reaching. In India, Bangalore is now considered South Asia's "Silicon Valley." Foreign software investment in India has grown from $150 million in 1992 to $2 billion in 1996. The first reason is a change in government tax and ownership regulations. The second big reason is the technical talent. After the United States and Japan, India has the largest scientific and technical pool of talent. And it is growing. The best and the brightest who use to head to the United States are starting to return or just plain stay put. What's more, salaries for software engineers are rising along with opportunities to do advance technical work.

Suraj Shenoy, 24, works at Digital House in Bangalore, India. He moved from Freehold, New Jersey, USA, back to India for the cosmopolitan lifestyle, and economic and career opportunities in Bangalore. "If you're young and moving up, this place is tops," Shenoy states (*The Star-Ledger*, 4 Aug. '97).

With annual sales of 2.1 million personal computers for 1996, China's market has become number two in Asia passing South Korea, which previously was number two. (Japan remains number one in the region.) This growth spurt for personal computers and peripherals is reflected also in publication sales. Computer publications in China have a combined readership of 18 million. This is a bigger circulation than China's *People's Daily*. IDG (International Data Group), in partnership with the Chinese government, puts out this highly sought information. *China Computerworld* is their flagship moneymaker with 70% of each issue's pages carrying advertisements (source: *Forbes*, 25 Aug. 97). These kind of numbers show the still-growing market for computers and their many related products. This, in turn, shows why the hiring market is so strong.

However, a cornerstone to this job market is "virtual manufacturing." Manufacturers who represented 51 percent of the companies offering computer engineering graduate jobs are trying to take expensive guesswork out of the picture. CAD (computer-aided design) allows designers for manufacturers to fine tune their ideas. Thus, the likelihood that the dreamed up product and procedure will succeed in a reality-based environment are higher with less cost and frustration.

However, more importantly, huge amounts of data now can be easily stored and transferred around the world. This way many designers as well as many computer programs can all be working on a given project. When Boeing engineers designed the 777 jetliner, they used a software system called CATIA developed by DASSAULT Systèmes in France.

There are even systems, such as DMAPS by DASSAULT and a suite of programs from Tecnomatix Technologies based in Israel, that can simulate entire factories. The goal is to be able to computer model the entire production process (source: *The Economist*).

Shifting back to job opportunities, the continuing resurgence of the manufacturing sector's portion of new grad offers hit 36 percent in 1997. (For this decade, 1990 was the high with 45.3% of the job offers for new grads coming from the manufacturing sector. However, it then proceeded to steadily dip to 28.8% for 1994 before again going up.)

©Image Bank/P. Ridenour/© 1996 PhotoDisc, Inc.

According to NACE, the 1997-1998 recruitment cycle should be a financially rewarding replay of last year. Technology and computer-related disciplines should continue to enjoy multiple offers, signing bonuses and the limelight. This is all mainly thanks to the manufacturing sector- -the engine that drives the economy- as it continues its comeback.

—MKC

Ergonomics

It's 3:00 a.m. as you pull another all-nighter to finish a research paper. You're getting your umpteenth cup of coffee when you realize that your wrists feel tight and sore. There's also a slight tingling and coldness in your hands.

Dawn breaks and your paper is finally done. After a little shut-eye, you decide to give yourself a mental break; so you surf the Web. But you discover that annoying soreness in your fingers and wrists is still there. Even your neck, back and shoulders are still bothersome.

What is it?

If this sounds familiar, you're probably suffering from the early symptoms of a Repetitive Stress Injury (RSI). According to Dr. Emil Pascarelli (from his book, *Repetitive Strain Injury: A Computer User's Guide*, Wiley, 1992), an RSI is defined as "a cumulative trauma disorder stemming from prolonged repetitive, forceful or awkward hand movements," e.g., strenuous keyboard use.

Poor posture and typing habits, ill-adapted furniture and keyboards, and a fast-paced, heavy workload are all contributing factors to an RSI. The result, Pascarelli states, is "damage to the muscles, tendons and nerves of the neck, shoulder, forearm and hand which can cause pain, weakness, numbness or impairment of motor control."

But if you stay away from the computer for a couple of days, your wrists, hands and fingers will start to feel better. So why worry?

Well, a few days sans keyboard use will ease the discomfort somewhat. But if your typing habits remain the same, you're setting yourself up for some potentially serious problems. RSI in its worst form can involve tightness or stiffness in hands and/or wrists, tingling or numbness in fingers, the need to constantly massage your hands/wrists, hypersensitivity in hands after minimal use, loss of coordination and pain. This can prevent you from turning a door knob, holding a telephone or even holding a newspaper—let alone using a keyboard.

RSI is a preventable condition. Even if you're only experiencing a few symptoms, or if they go away easily, you are at risk. Ignoring this problem can lead to severe, long-term disability.

What can you do to prevent an RSI? Change your work habits. Pay attention to your body. When something tingles or hurts, stop and rest. Structure your workload so that you're not forcing yourself to work for long, uninterrupted sessions. Take a break—many of them. And adopt some simple practices.

Sit up straight

Your mother was right; you should sit up straight. Maintaining proper alignment of the spine is the cornerstone of RSI preventative measures.

Slouching or slumping in your chair compresses the spine which can lead to lower back pain and/or neck and shoulder stiffness and discomfort. To correctly align your spine, keep your head up with your "ears in line with the shoulders and hips . . . your shoulders should hold your chest open and your arm supporting your hands over the keyboard" (*Repetitive Strain Injury*, Wiley, 1992).

Get your hands up

Poor posture contributes to improper arm and wrist positioning. Slouchers tend to rest their elbows and wrists on the work surface while typing. This forces the tendons in the wrists to bend in awkward and unnatural positions that can lead to nerve damage.

Position the wrists above the keyboard in a slightly arched position. (So they are raised higher than the hands.) They should not rest on anything (especially wrist rests) while typing.

The wrists should also be held in a stationary and neutral position. Pivoting at the wrist to reach a key puts strain on the tendons. The entire arm should be used to guide the fingers across the keyboard.

A light touch

Many computer users pound the keys as they type. Pressure and anxiety due to deadlines, workload and job/academic performance can lead to this behavior. But taking out your frustrations on the keyboard will only cause strain or injury.

Pace yourself. Realize that you will not finish the work any faster if you slam your fingers down on the keyboard. A light touch is just as effective.

And stay off the mouse as much as possible. Using it is conducive to wrist pivoting.

Take five

Take a five- to ten-minute break every hour or so. Not only will this allow you to stretch your muscles (very important), it will prompt you to reassess your posture and wrist/arm/hand positions. Even if you don't have the time to get up, mini-stretch breaks at your desk will also help.

Proper setup

A user-friendly workstation can also help you maintain these practices. Any one of the following can help keep your posture in good shape:

- Position your monitor at eye level (or close to it) and keep it directly above the keyboard.
- Make sure your keyboard is at a level so that your arms are bent at a 45° angle.
- Use a chair that has good lower back support.
- Use a footrest to help alleviate pressure on your spine.

Additional help

An ergonomic aid such as a split keyboard may be helpful. It is designed to keep your hands in a more natural position while typing. There are also monitoring programs designed to remind the user to take periodic breaks during a work session. Voice-activated software allows the user to dictate to the computer instead of typing.

While helpful, these aids should not be the only preventative methods taken. According to Paul Marxhausen, RSI sufferer and author of the Computer Related Repetitive Strain Injury website (http://www.engr.unl.edu/ee/eeshop/rsi.html), "correct typing technique and posture, the right equipment setup, and good work habits are much more important for prevention" than ergonomic aids.

FYI

For further information on RSIs, how to prevent them and treatment options, we recommend starting with Marxhausen's web site. It is informative and provides excellent references to other material on the subject.

— Lisa Dayne