```
                        else
                                transition(s,t,j,i);
                    }
    } while (!stuck);
}
```

When does local search do well?

- *When there is great coherence in the solution space* – Hill climbing is at its best when the solution space is *convex*. In other words, it consists of exactly one hill. No matter where you start on the hill, there is always a direction to walk up until you are at the absolute global maximum.

  Many natural problems do have this property. We can think of a binary search as starting in the middle of a search space, where exactly one of the two possible directions we can walk will get us closer to the target key. The simplex algorithm for linear programming (see Section 13.6 (page 411)) is nothing more than hill-climbing over the right solution space, yet guarantees us the optimal solution to any linear programming problem.

- *Whenever the cost of incremental evaluation is much cheaper than global evaluation* – It costs $\Theta(n)$ to evaluate the cost of an arbitrary n-vertex candidate TSP solution, because we must total the cost of each edge in the circular permutation describing the tour. Once that is found, however, the cost of the tour after swapping a given pair of vertices can be determined in constant time.

  If we are given a very large value of n and a very small budget of how much time we can spend searching, we are better off using it to do several incremental evaluations than a few random samples, even if we are looking for a needle in a haystack.

The primary drawback of a local search is that soon there isn't anything left for us to do as we find the local optimum. Sure, if we have more time we could start from different random points, but in landscapes of many low hills we are unlikely to stumble on the optimum.

How does local search do on TSP? Much better than random sampling for a similar amount of time. With over a total of 1.5 million tour evaluations in our 48-city TSP instance, our best local search tour had a length of 40,121.2—only 19.6% more than the optimal tour of 33,523.7.

This is good, but not great. You would not be happy to learn you are paying 19.6% more taxes than you should. Figure 7.9 illustrates the trajectory of a local search: repeated streaks from random tours down to decent solutions of fairly similar quality. We need more powerful methods to get closer to the optimal solution.

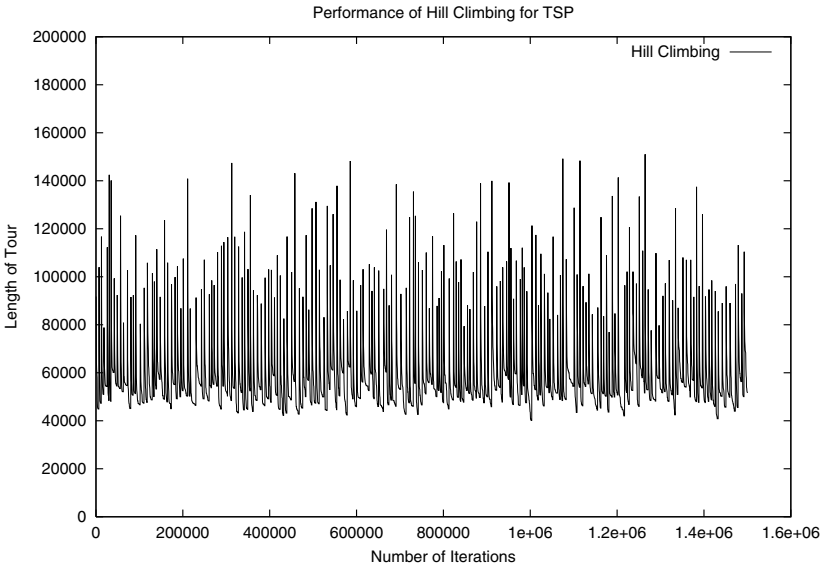

Figure 7.9: Search time/quality tradeoffs for TSP using hill climbing.

7.5.3 Simulated Annealing

Simulated annealing is a heuristic search procedure that allows occasional transitions leading to more expensive (and hence inferior) solutions. This may not sound like progress, but it helps keep our search from getting stuck in local optima. That poor fellow trapped on the top floor of a ski lodge would do better to break the glass and jump out the window if he really wanted to reach the top of the mountain.

The inspiration for simulated annealing comes from the physical process of cooling molten materials down to the solid state. In thermodynamic theory, the energy state of a system is described by the energy state of each particle constituting it. A particle's energy state jumps about randomly, with such transitions governed by the temperature of the system. In particular, the transition probability $P(e_i, e_j, T)$ from energy e_i to e_j at temperature T is given by

$$P(e_i, e_j, T) = e^{(e_i - e_j)/(k_B T)}$$

where k_B is a constant—called Boltzmann's constant.

What does this formula mean? Consider the value of the exponent under different conditions. The probability of moving from a high-energy state to a lower-energy state is very high. But, there is still a nonzero probability of accepting a

transition into a high-energy state, with such small jumps much more likely than big ones. The higher the temperature, the more likely energy jumps will occur.

```
Simulated-Annealing()
    Create initial solution S
    Initialize temperature t
    repeat
        for i = 1 to iteration-length do
            Generate a random transition from S to S_i
            If (C(S) ≥ C(S_i)) then S = S_i
            else if (e^((C(S)−C(S_i))/(k·t)) > random[0,1)) then S = S_i
        Reduce temperature t
    until (no change in C(S))
    Return S
```

What relevance does this have for combinatorial optimization? A physical system, as it cools, seeks to reach a minimum-energy state. Minimizing the total energy is a combinatorial optimization problem for any set of discrete particles. Through random transitions generated according to the given probability distribution, we can mimic the physics to solve arbitrary combinatorial optimization problems.

Take-Home Lesson: Forget about this molten metal business. Simulated annealing is effective because it spends much more of its time working on good elements of the solution space than on bad ones, and because it avoids getting trapped repeatedly in the same local optima.

As with a local search, the problem representation includes both a representation of the solution space and an easily computable cost function $C(s)$ measuring the quality of a given solution. The new component is the *cooling schedule*, whose parameters govern how likely we are to accept a bad transition as a function of time.

At the beginning of the search, we are eager to use randomness to explore the search space widely, so the probability of accepting a negative transition should be high. As the search progresses, we seek to limit transitions to local improvements and optimizations. The cooling schedule can be regulated by the following parameters:

- *Initial system temperature* – Typically $t_1 = 1$.
- *Temperature decrement function* – Typically $t_k = \alpha \cdot t_{k-1}$, where $0.8 \leq \alpha \leq 0.99$. This implies an exponential decay in the temperature, as opposed to a linear decay.
- *Number of iterations between temperature change* – Typically, 100 to 1,000 iterations might be permitted before lowering the temperature.

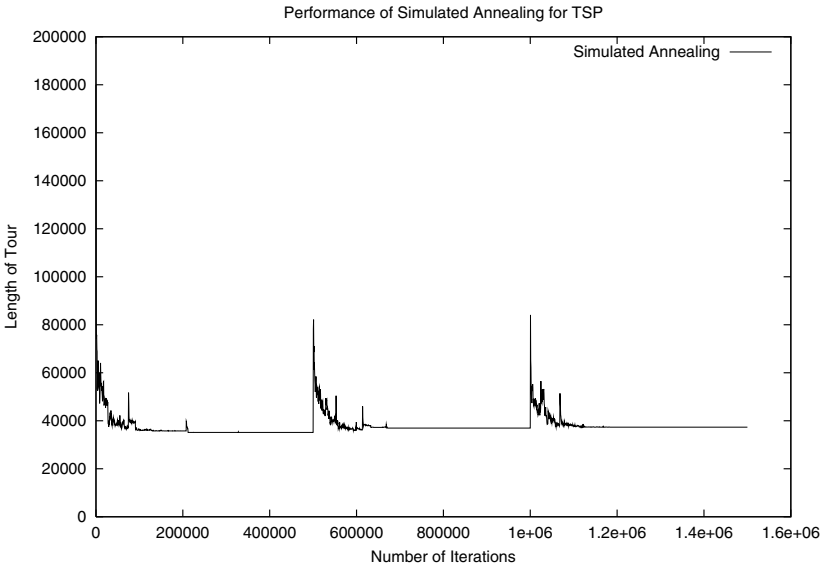

Figure 7.10: Search time/quality tradeoffs for TSP using simulated annealing

- *Acceptance criteria* – A typical criterion is to accept any transition from s_i to s_{i+1} when $C(s_{i+1}) < C(s_i)$, and also accept a negative transition whenever

  $$e^{-\frac{(C(s_i)-C(s_{i+1}))}{k \cdot t_i}} \geq r,$$

  where r is a random number $0 \leq r < 1$. The constant k normalizes this cost function so that almost all transitions are accepted at the starting temperature.

- *Stop criteria* – Typically, when the value of the current solution has not changed or improved within the last iteration or so, the search is terminated and the current solution reported.

Creating the proper cooling schedule is somewhat of a trial-and-error process of mucking with constants and seeing what happens. It probably pays to start from an existing implementation of simulated annealing, so check out my full implementation (at *http://www.algorist.com*) as well as others provided in Section 13.5 (page 407).

Compare the search/time execution profiles of our three heuristics. The cloud of points corresponding to random sampling is significantly worse than the solutions

encountered by the other heuristics. The scores of the short streaks corresponding to runs of hill-climbing solutions are clearly much better.

But best of all are the profiles of the three simulated annealing runs in Figure 7.10. All three runs lead to much better solutions than the best hill-climbing result. Further, it takes relatively few iterations to score most of the improvement, as shown by the three rapid plunges toward optimum we see with simulated annealing.

Using the same 1,500,000 iterations as the other methods, simulated annealing gave us a solution of cost 36,617.4—only 9.2% over the optimum. Even better solutions are available to those willing to wait a few minutes. Letting it run for 5,000,000 iterations got the score down to 34,254.9, or 2.2% over the optimum. There were no further improvements after I cranked it up to 10,000,000 iterations.

In expert hands, the best problem-specific heuristics for TSP can slightly outperform simulated annealing. But the simulated annealing solution works admirably. It is my heuristic method of choice.

Implementation

The implementation follows the pseudocode quite closely:

```
anneal(tsp_instance *t, tsp_solution *s)
{
  int i1, i2;                  /* pair of items to swap */
  int i,j;                     /* counters */
  double temperature;          /* the current system temp */
  double current_value;        /* value of current state */
  double start_value;          /* value at start of loop */
  double delta;                /* value after swap */
  double merit, flip;          /* hold swap accept conditions*/
  double exponent;             /* exponent for energy funct*/
  double random_float();
  double solution_cost(), transition();

  temperature = INITIAL_TEMPERATURE;

  initialize_solution(t->n,s);
  current_value = solution_cost(s,t);

  for (i=1; i<=COOLING_STEPS; i++) {
          temperature *= COOLING_FRACTION;

          start_value = current_value;

          for (j=1; j<=STEPS_PER_TEMP; j++) {
```

```
            /* pick indices of elements to swap */
            i1 = random_int(1,t->n);
            i2 = random_int(1,t->n);

            flip = random_float(0,1);

            delta = transition(s,t,i1,i2);
            exponent = (-delta/current_value)/(K*temperature);
            merit = pow(E,exponent);

            if (delta < 0)                              /*ACCEPT-WIN*/
                 current_value = current_value+delta;
            else { if (merit > flip)                    /*ACCEPT-LOSS*/
                 current_value = current_value+delta;
            else                                        /* REJECT */
                 transition(s,t,i1,i2);
            }
          }

          /* restore temperature if progress has been made */
          if ((current_value-start_value) < 0.0)
                 temperature = temperature/COOLING_FRACTION;
  }
}
```

7.5.4 Applications of Simulated Annealing

We provide several examples to demonstrate how these components can lead to elegant simulated annealing solutions for real combinatorial search problems.

Maximum Cut

The "maximum cut" problem seeks to partition the vertices of a weighted graph G into sets V_1 and V_2 to maximize the weight (or number) of edges with one vertex in each set. For graphs that specify an electronic circuit, the maximum cut in the graph defines the largest amount of data communication that can take place in the circuit simultaneously. As discussed in Section 16.6 (page 541), maximum cut is NP-complete.

How can we formulate maximum cut for simulated annealing? The solution space consists of all 2^{n-1} possible vertex partitions. We save a factor of two over all vertex subsets because vertex v_1 can be assumed to be fixed on the left side of the partition. The subset of vertices accompanying it can be represented using

a bit vector. The cost of a solution is the sum of the weights cut in the current configuration. A natural transition mechanism selects one vertex at random and moves it across the partition simply by flipping the corresponding bit in the bit vector. The change in the cost function will be the weight of its old neighbors minus the weight of its new neighbors. This can be computed in time proportional to the degree of the vertex.

This kind of simple, natural modeling is the right type of heuristic to seek in practice.

Independent Set

An "independent set" of a graph G is a subset of vertices S such that there is no edge with both endpoints in S. The maximum independent set of a graph is the largest such empty induced subgraph. Finding large independent sets arises in dispersion problems associated with facility location and coding theory, as discussed in Section 16.2 (page 528).

The natural state space for a simulated annealing solution would be all 2^n subsets of the vertices, represented as a bit vector. As with maximum cut, a simple transition mechanism would add or delete one vertex from S.

One natural cost function for subset S might be 0 if S contains an edge, and $|S|$ if it is indeed an independent set. This function ensures that we work towards an independent set at all times. However, this condition is strict enough that we are liable to move only in a narrow portion of the possible search space. More flexibility in the search space and quicker cost function computations can result from allowing nonempty graphs at the early stages of cooling. Better in practice would be a cost function like $C(S) = |S| - \lambda \cdot m_S/T$, where λ is a constant, T is the temperature, and m_S is the number of edges in the subgraph induced by S. The dependence of $C(S)$ on T ensures that the search will drive the edges out faster as the system cools.

Circuit Board Placement

In designing printed circuit boards, we are faced with the problem of positioning modules (typically, integrated circuits) on the board. Desired criteria in a layout may include (1) minimizing the area or aspect ratio of the board so that it properly fits within the allotted space, and (2) minimizing the total or longest wire length in connecting the components. Circuit board placement is representative of the kind of messy, multicriterion optimization problems for which simulated annealing is ideally suited.

Formally, we are given a collection of rectangular modules $r_1, \ldots, r_n$, each with associated dimensions $h_i \times l_i$. Further, for each pair of modules r_i, r_j, we are given the number of wires w_{ij} that must connect the two modules. We seek a placement of the rectangles that minimizes area and wire length, subject to the constraint that no two rectangles overlap each other.

The state space for this problem must describe the positions of each rectangle. To make this discrete, the rectangles can be restricted to lie on vertices of an integer grid. Reasonable transition mechanisms including moving one rectangle to a different location, or swapping the position of two rectangles. A natural cost function would be

$$C(S) = \lambda_{area}(S_{height} \cdot S_{width}) + \sum_{i=1}^{n}\sum_{j=1}^{n}(\lambda_{wire} \cdot w_{ij} \cdot d_{ij} + \lambda_{overlap}(r_i \cap r_j))$$

where λ_{area}, λ_{wire}, and $\lambda_{overlap}$ are constants governing the impact of these components on the cost function. Presumably, $\lambda_{overlap}$ should be an inverse function of temperature, so after gross placement it adjusts the rectangle positions to be disjointed.

Take-Home Lesson: Simulated annealing is a simple but effective technique for efficiently obtaining good but not optimal solutions to combinatorial search problems.

7.6 War Story: Only it is Not a Radio

"Think of it as a radio," he chuckled. "Only it is not a radio."

I'd been whisked by corporate jet to the research center of a large but very secretive company located somewhere east of California. They were so paranoid that I never got to see the object we were working on, but the people who brought me in did a great job of abstracting the problem.

The application concerned a manufacturing technique known as *selective assembly*. Eli Whitney started the Industrial Revolution through his system of *interchangeable parts*. He carefully specified the manufacturing tolerances on each part in his machine so that the parts were *interchangeable*, meaning that any legal cog-widget could be used to replace any other legal cog-widget. This greatly sped up the process of manufacturing, because the workers could just put parts together instead of having to stop to file down rough edges and the like. It made replacing broken parts a snap. This was a very good thing.

Unfortunately, it also resulted in large piles of cog-widgets that were slightly outside the manufacturing tolerance and thus had to be discarded. Another clever fellow then observed that maybe one of these defective cog-widgets could be used when all the *other* parts in the given assembly *exceeded* their required manufacturing tolerances. Good plus bad could well equal good enough. This is the idea of *selective assembly*.

"Each not-radio is made up of n different types of not-radio parts," he told me. For the ith part type (say the right flange gasket), we have a pile of s_i instances of this part type. Each part (flange gasket) comes with a measure of the degree to which it deviates from perfection. We need to match up the parts so as to create the greatest number of working not-radios as possible."

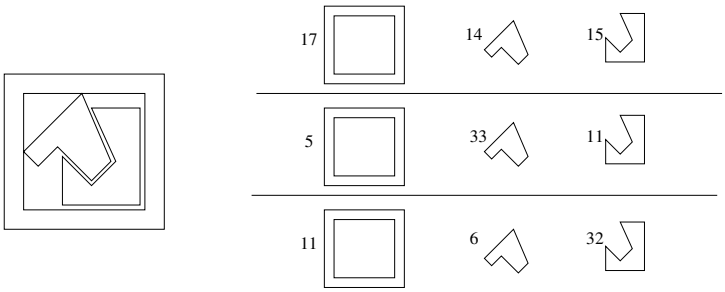

Figure 7.11: Part assignments for three not-radios, such that each had at most 50 points total defect

The situation is illustrated in Figure 7.11. Each not-radio consists of three parts, and the sum of the defects in any functional not-radio must total at most 50. By cleverly balancing the good and bad parts in each machine, we can use all the parts and make three working not-radios.

I thought about the problem. The simplest procedure would take the best part for each part type, make a not-radio out of them, and repeat until the not-radio didn't play (or do whatever a not-radio does). But this would create a small number of not-radios drastically varying in quality, whereas they wanted as many decent not-radios as possible.

The goal was to match up good parts and bad parts so the total amount of badness wasn't so bad. Indeed, the problem sounded related to *matching* in graphs (see Section 15.6 (page 498)). Suppose we built a graph where the vertices were the part instances, and add an edge for all two part instances that were within the total error tolerance. In graph matching, we seek the largest number of edges such that no vertex appears more than once in the matching. This is analogous to the largest number of two-part assemblies we can form from the given set of parts.

"I can solve your problem using matching," I announced, "Provided not-radios are each made of only two parts."

There was silence. Then they all started laughing at me. "*Everyone* knows not-radios have more than two parts," they said, shaking their heads.

That spelled the end of this algorithmic approach. Extending to more than two parts turned the problem into matching on hypergraphs,[2]—a problem which is NP-complete. Further, it might take exponential time in the number of part types just to build the graph, since we had to explicitly construct each possible hyperedge/assembly.

[2]A *hypergraph* is made up of edges that can contain more than two vertices each. They can be thought of as general collections of subsets of vertices/elements.

I went back to the drawing board. They wanted to put parts into assemblies so that no assembly would have more total defects than allowed. Described that way, it sounded like a packing problem. In the *bin packing* problem (see Section 17.9 (page 595)), we are given a collection of items of different sizes and asked to store them using the smallest possible number of bins, each of which has a fixed capacity of size k. Here, the assemblies represented the bins, each of which could absorb total defect $\leq k$. The items to pack represented the individual parts, whose size would reflect its quality of manufacture.

It wasn't pure bin packing, however, since parts came in different types. The application imposed constraints on the allowable contents of each bin. Creating the maximum number of not-radios meant that we sought a packing that maximized the number of bins which contained exactly one part for each of the m different parts types.

Bin packing is NP-complete, but is a natural candidate for a heuristic search approach. The solution space consists of assignments of parts to bins. We initially pick a random part of each type for each bin to give us a starting configuration for the search.

The local neighborhood operation involves moving parts around from one bin to another. We could move one part at a time, but more effective was *swapping* parts of a particular type between two randomly chosen bins. In such a swap, both bins remain complete not-radios, hopefully with better error tolerance than before. Thus, our swap operator required three random integers—one to select the appropriate parts type (from 1 to m) and two more to select the assembly bins involved (say from 1 to b).

The key decision was the cost function to use. They supplied the hard limit k on the total defect level for each *individual* assembly. But what was the best way to score a *set* of assemblies? We could just return the number of acceptable complete assemblies as our score—an integer from 1 to b. Although this was indeed what we wanted to optimize, it would not be sensitive to detect when we were making partial progress towards a solution. Suppose one of our swaps succeeded in bringing one of the nonfunctional assemblies much closer to the not-radio limit k. That would be a better starting point for further progress than the original, and should be favored.

My final cost function was as follows. I gave one point for every working assembly, and a significantly smaller total for each nonworking assembly based on how close it was to the threshold k. The score for a nonworking assembly decreased exponentially based on how much it was over k. Thus the optimizer would seek to maximize the number of working assemblies, and then try to drive another assembly close to the limit.

I implemented this algorithm, and then ran the search on the test case they provided. It was an instance taken directly from the factory floor. Not-radios turn out to contain $m = 8$ important parts types. Some parts types are more expensive than others, and so they have fewer available candidates to consider. The most

prefix \ suffix	AA	AG	GA	GG
AA	$AAAA$	$AAAG$	$AAGA$	$AAGG$
AG	$AGAA$	$AGAG$	$AGGA$	$AGGG$
GA	$GAAG$	$GAAG$	$GAGA$	$GAGG$
GG	$GGAA$	$GGAG$	$GGGA$	$GGGG$

Figure 7.12: A prefix-suffix array of all purine 4-mers

constrained parts type had only eight representatives, so there could be at most eight possible assemblies from this given mix.

I watched as simulated annealing chugged and bubbled on the problem instance. The number of completed assemblies instantly climbed (1, 2, 3, 4) before progress started to slow a bit. Then came 5 and 6 in a hiccup, with a pause before assembly 7 came triumphantly together. But tried as it might, the program could not put together eight not-radios before I lost interest in watching.

I called and tried to admit defeat, but they wouldn't hear it. It turns out that the best the factory had managed after extensive efforts was only *six* working not-radios, so my result represented a significant improvement!

7.7 War Story: Annealing Arrays

The war story of Section 3.9 (page 94) reported how we used advanced data structures to simulate a new method for sequencing DNA. Our method, interactive sequencing by hybridization (SBH), required building arrays of specific oligonucleotides on demand.

A biochemist at Oxford University got interested in our technique, and moreover he had in his laboratory the equipment we needed to test it out. The Southern Array Maker, manufactured by Beckman Instruments, prepared discrete oligonucleotide sequences in 64 parallel rows across a polypropylene substrate. The device constructs arrays by appending single characters to each cell along specific rows and columns of arrays. Figure 7.12 shows how to construct an array of all $2^4 = 16$ purine (A or G) 4-mers by building the prefixes along rows and the suffixes along columns. This technology provided an ideal environment for testing the feasibility of interactive SBH in a laboratory, because with proper programming it gave a way to fabricate a wide variety of oligonucleotide arrays on demand.

However, we had to provide the proper programming. Fabricating complicated arrays required solving a difficult combinatorial problem. We were given as input a set of n strings (representing oligonucleotides) to fabricate in an $m \times m$ array (where $m = 64$ on the Southern apparatus). We had to produce a schedule of row and column commands to realize the set of strings S. We proved that the

problem of designing dense arrays was NP-complete, but that didn't really matter. My student Ricky Bradley and I had to solve it anyway.

"We are going to have to use a heuristic," I told him. "So how can we model this problem?"

"Well, each string can be partitioned into prefix and suffix pairs that realize it. For example, the string ACC can be realized in four different ways: prefix '' and suffix ACC, prefix A and suffix CC, prefix AC and suffix C, or prefix ACC and suffix ''. We seek the smallest set of prefixes and suffixes that together realize all the given strings," Ricky said.

"Good. This gives us a natural representation for simulated annealing. The state space will consist of all possible subsets of prefixes and suffixes. The natural transitions between states might include inserting or deleting strings from our subsets, or swapping a pair in or out."

"What's a good cost function?" he asked.

"Well, we need as small an array as possible that covers all the strings. How about taking the maximum of number of rows (prefixes) or columns (suffixes) used in our array, plus the number of strings from S that are not yet covered. Try it and let's see what happens."

Ricky went off and implemented a simulated annealing program along these lines. It printed out the state of the solution each time a transition was accepted and was fun to watch. The program quickly kicked out unnecessary prefixes and suffixes, and the array began shrinking rapidly in size. But after several hundred iterations, progress started to slow. A transition would knock out an unnecessary suffix, wait a while, then add a different suffix back again. After a few thousand iterations, no real improvement was happening.

"The program doesn't seem to recognize when it is making progress. The evaluation function only gives credit for minimizing the larger of the two dimensions. Why not add a term to give some credit to the other dimension."

Ricky changed the evaluation function, and we tried again. This time, the program did not hesitate to improve the shorter dimension. Indeed, our arrays started to be skinny rectangles instead of squares.

"OK. Let's add another term to the evaluation function to give it points for being roughly square."

Ricky tried again. Now the arrays were the right shape, and progress was in the right direction. But the progress was still slow.

"Too many of the insertion moves don't affect many strings. Maybe we should skew the random selections so that the important prefix/suffixes get picked more often."

Ricky tried again. Now it converged faster, but sometimes it still got stuck. We changed the cooling schedule. It did better, but was it doing well? Without a lower bound knowing how close we were to optimal, it couldn't really tell how good our solution was. We tweaked and tweaked until our program stopped improving.

Our final solution refined the initial array by applying the following random moves:

Figure 7.13: Compression of the HIV array by simulated annealing – after 0, 500, 1,000, and 5,750 iterations

---

- *swap* – swap a prefix/suffix on the array with one that isn't.
- *add* – add a random prefix/suffix to the array.
- *delete* – delete a random prefix/suffix from the array.
- *useful add* – add the prefix/suffix with the highest usefulness to the array.
- *useful delete* – delete the prefix/suffix with the lowest usefulness from the array.
- *string add* – randomly select a string not on the array, and add the most useful prefix and/or suffix to cover this string.

A standard cooling schedule was used, with an exponentially decreasing temperature (dependent upon the problem size) and a temperature-dependent Boltzmann criterion for accepting states that have higher costs. Our final cost function was defined as

$$cost = 2 \times max + min + \frac{(max - min)^2}{4} + 4(str_{total} - str_{in})$$

where max is the size of the maximum chip dimension, min is the size of the minimum chip dimension, $str_{total} = |S|$, and str_{in} is the number of strings from S currently on the chip.

How well did we do? Figure 7.13 shows the convergence of a custom array consisting of the 5,716 unique 7-mers of the HIV virus. Figure 7.13 shows snapshots of the state of the chip at four points during the annealing process (0, 500, 1,000, and the final chip at 5,750 iterations). Black pixels represent the first occurrence of an HIV 7-mer. The final chip size here is 130×132—quite an improvement over

the initial size of 192×192. It took about fifteen minutes' worth of computation to complete the optimization, which was perfectly acceptable for the application.

But how well did we do? Since simulated annealing is only a heuristic, we really don't know how close to optimal our solution is. I think we did pretty well, but can't really be sure. Simulated annealing is a good way to handle complex optimization problems. However, to get the best results, expect to spend more time tweaking and refining your program than you did in writing it in the first place. This is dirty work, but sometimes you have to do it.

7.8 Other Heuristic Search Methods

Several heuristic search methods have been proposed to search for good solutions for combinatorial optimization problems. Like simulated annealing, many techniques relies on analogies to real-world physical processes. Popular methods include *genetic algorithms*, *neural networks*, and *ant colony optimization*.

The intuition behind these methods is highly appealing, but skeptics decry them as voodoo optimization techniques that rely more on nice analogies to nature than demonstrated computational results on problems that have been studied using other methods.

The question isn't whether you can get decent answers for many problems given enough effort using these techniques. Clearly you can. The real question is whether they lead to *better* solutions with *less implementation complexity* than the other methods we have discussed.

In general, I don't believe that they do. But in the spirit of free inquiry, I introduce genetic algorithms, which is the most popular of these methods. See the chapter notes for more detailed readings.

Genetic Algorithms

Genetic algorithms draw their inspiration from evolution and natural selection. Through the process of natural selection, organisms adapt to optimize their chances for survival in a given environment. Random mutations occur in an organism's genetic description, which then get passed on to its children. Should a mutation prove helpful, these children are more likely to survive and reproduce. Should it be harmful, these children won't, and so the bad trait will die with them.

Genetic algorithms maintain a "population" of solution candidates for the given problem. Elements are drawn at random from this population and allowed to "reproduce" by combining aspects of the two-parent solutions. The probability that an element is chosen to reproduce is based on its "fitness,"—essentially the cost of the solution it represents. Unfit elements die from the population, to be replaced by a successful-solution offspring.

The idea behind genetic algorithms is extremely appealing. However, they don't seem to work as well on practical combinatorial optimization problems as simulated

annealing does. There are two primary reasons for this. First, it is quite unnatural to model applications in terms of genetic operators like mutation and crossover on bit strings. The pseudobiology adds another level of complexity between you and your problem. Second, genetic algorithms take a very long time on nontrivial problems. The crossover and mutation operations typically make no use of problem-specific structure, so most transitions lead to inferior solutions, and convergence is slow. Indeed, the analogy with evolution—where significant progress require millions of years—can be quite appropriate.

We will not discuss genetic algorithms further, to discourage you from considering them for your applications. However, pointers to implementations of genetic algorithms are provided in Section 13.5 (page 407) if you really insist on playing with them.

> *Take-Home Lesson:* I have *never* encountered any problem where genetic algorithms seemed to me the right way to attack it. Further, I have *never* seen any computational results reported using genetic algorithms that have favorably impressed me. Stick to simulated annealing for your heuristic search voodoo needs.

7.9 Parallel Algorithms

Two heads are better than one, and more generally, n heads are better than $n-1$. Parallel processing is becoming more important with the advent of cluster computing and multicore processors. It seems like the easy way out of hard problems. Indeed, sometimes, for some problems, parallel algorithms are the most effective solution. High-resolution, real-time graphics applications must render thirty frames per second for realistic animation. Assigning each frame to a distinct processor, or dividing each image into regions assigned to different processors, might be the only way to get the job done in time. Large systems of linear equations for scientific applications are routinely solved in parallel.

However, there are several pitfalls associated with parallel algorithms that you should be aware of:

- *There is often a small upper bound on the potential win* – Suppose that you have access to twenty processors that can be devoted exclusively to your job. Potentially, these could be used to speed up the fastest sequential program by up to a factor of twenty. That is nice, but greater performance gains maybe possible by finding a better sequential algorithm. Your time spent parallelizing a code might well be better spent enhancing the sequential version. Performance-tuning tools such as profilers are better developed for sequential machines than for parallel models.

- *Speedup means nothing* – Suppose my parallel program runs 20 times faster on a 20-processor machine than it does on one processor. That's great, isn't

it? If you always get linear speedup and have an arbitrary number of processors, you will eventually beat any sequential algorithm. However, a carefully designed sequential algorithm can often beat an easily-parallelized code running on a typical parallel machine. The one-processor parallel version of your code is likely to be a crummy sequential algorithm, so measuring speedup typically provides an unfair test of the benefits of parallelism.

The classic example of this occurs in the minimax game-tree search algorithm used in computer chess programs. A brute-force tree search is embarrassingly easy to parallelize: just put each subtree on a different processor. However, a lot of work gets wasted because the same positions get considered on different machines. Moving from a brute-force search to the more clever alpha-beta pruning algorithm can easily save 99.99% of the work, thus dwarfing any benefits of a parallel brute-force search. Alpha-beta can be parallelized, but not easily, and the speedups grow surprisingly slowly as a function of the number of processors you have.

- *Parallel algorithms are tough to debug* – Unless your problem can be decomposed into several independent jobs, the different processors must communicate with each other to end up with the correct final result. Unfortunately, the nondeterministic nature of this communication makes parallel programs notoriously difficult to debug. Perhaps the best example is *Deep Blue*—the world-champion chess computer. Although it eventually beat Kasparov, over the years it lost several games in embarrassing fashion due to bugs, mostly associated with its extensive parallelism.

I recommend considering parallel processing only after attempts at solving a problem sequentially prove too slow. Even then, I would restrict attention to algorithms that parallelize the problem by partitioning the input into distinct tasks where no communication is needed between the processors, except to collect the final results. Such large-grain, naive parallelism can be simple enough to be both implementable and debuggable, because it really reduces to producing a good sequential implementation. There can be pitfalls even in this approach, however, as shown in the war story below.

7.10 War Story: Going Nowhere Fast

In Section 2.8 (page 51), I related our efforts to build a fast program to test Waring's conjecture for pyramidal numbers. At that point, my code was fast enough that it could complete the job in a few weeks running in the background of a desktop workstation. This option did not appeal to my supercomputing colleague, however.

"Why don't we do it in parallel?" he suggested. "After all, you have an outer loop doing the same calculation on each integer from 1 to 1,000,000,000. I can split this range of numbers into different intervals and run each range on a different processor. Watch, it will be easy."

He set to work trying to do our computations on an Intel IPSC-860 hypercube using 32 nodes with 16 megabytes of memory per node—very big iron for the time. However, instead of getting answers, I was treated to a regular stream of e-mail about system reliability over the next few weeks:

- "Our code is running fine, except one processor died last night. I will rerun."
- "This time the machine was rebooted by accident, so our long-standing job was killed."
- "We have another problem. The policy on using our machine is that nobody can command the entire machine for more than thirteen hours, under any condition."

Still, eventually, he rose to the challenge. Waiting until the machine was stable, he locked out 16 processors (half the computer), divided the integers from 1 to 1,000,000,000 into 16 equal-sized intervals, and ran each interval on its own processor. He spent the next day fending off angry users who couldn't get their work done because of our rogue job. The instant the first processor completed analyzing the numbers from 1 to 62,500,000, he announced to all the people yelling at him that the rest of the processors would soon follow.

But they didn't. He failed to realize that the time to test each integer increased as the numbers got larger. After all, it would take longer to test whether 1,000,000,000 could be expressed as the sum of three pyramidal numbers than it would for 100. Thus, at slower and slower intervals each new processor would announce its completion. Because of the architecture of the hypercube, he couldn't return any of the processors until our entire job was completed. Eventually, half the machine and most of its users were held hostage by one, final interval.

What conclusions can be drawn from this? If you are going to parallelize a problem, be sure to balance the load carefully among the processors. Proper load balancing, using either back-of-the-envelope calculations or the partition algorithm we will develop in Section 8.5 (page 294), would have significantly reduced the time we needed the machine, and his exposure to the wrath of his colleagues.

Chapter Notes

The treatment of backtracking here is partially based on my book *Programming Challenges* [SR03]. In particular, the `backtrack` routine presented here is a generalization of the version in Chapter 8 of [SR03]. Look there for my solution to the famous *eight queens problem*, which seeks all chessboard configurations of eight mutually nonattacking queens on an 8×8 board.

The original paper on simulated annealing [KGV83] included an application to VLSI module placement problems. The applications from Section 7.5.4 (page 258) are based on material from [AK89].

The heuristic TSP solutions presented here employ vertex-swap as the local neighborhood operation. In fact, edge-swap is a more powerful operation. Each edge-swap changes two edges in the tour at most, as opposed to at most four edges with a vertex-swap. This improves the possibility of a local improvement. However, more sophisticated data structures are necessary to efficiently maintain the order of the resulting tour [FJMO93].

The different heuristic search techniques are ably presented in Aarts and Lenstra [AL97], which I strongly recommend for those interested in learning more about heuristic searches. Their coverage includes *tabu search*, a variant of simulated annealing that uses extra data structures to avoid transitions to recently visited states. Ant colony optimization is discussed in [DT04]. See [MF00] for a more favorable view of genetic algorithms and the like.

More details on our combinatorial search for optimal chessboard-covering positions appear in our paper [RHS89]. Our work using simulated annealing to compress DNA arrays was reported in [BS97]. See Pugh [Pug86] and Coullard et al. [CGJ98] for more on selective assembly. Our parallel computations on pyramidal numbers were reported in [DY94].

7.11 Exercises

Backtracking

7-1. *[3]* A *derangement* is a permutation p of $\{1, \ldots, n\}$ such that no item is in its proper position, i.e. $p_i \neq i$ for all $1 \leq i \leq n$. Write an efficient backtracking program with pruning that constructs all the derangements of n items.

7-2. *[4] Multisets* are allowed to have repeated elements. A multiset of n items may thus have fewer than $n!$ distinct permutations. For example, $\{1, 1, 2, 2\}$ has only six different permutations: $\{1, 1, 2, 2\}$, $\{1, 2, 1, 2\}$, $\{1, 2, 2, 1\}$, $\{2, 1, 1, 2\}$, $\{2, 1, 2, 1\}$, and $\{2, 2, 1, 1\}$. Design and implement an efficient algorithm for constructing all permutations of a multiset.

7-3. *[5]* Design and implement an algorithm for testing whether two graphs are isomorphic to each other. The graph isomorphism problem is discussed in Section 16.9 (page 550). With proper pruning, graphs on hundreds of vertices can be tested reliably.

7-4. *[5]* Anagrams are rearrangements of the letters of a word or phrase into a different word or phrase. Sometimes the results are quite striking. For example, "MANY VOTED BUSH RETIRED" is an anagram of "TUESDAY NOVEMBER THIRD," which correctly predicted the result of the 1992 U.S. presidential election. Design and implement an algorithm for finding anagrams using combinatorial search and a dictionary.

7-5. *[8]* Design and implement an algorithm for solving the subgraph isomorphism problem. Given graphs G and H, does there exist a subgraph H' of H such that G is isomorphic to H'? How does your program perform on such special cases of subgraph isomorphism as Hamiltonian cycle, clique, independent set, and graph isomorphism?

7-6. *[8]* In the turnpike reconstruction problem, you are given $n(n-1)/2$ distances in sorted order. The problem is to find the positions of n points on the line that give rise to these distances. For example, the distances $\{1, 2, 3, 4, 5, 6\}$ can be determined by placing the second point 1 unit from the first, the third point 3 from the second, and the fourth point 2 from the third. Design and implement an efficient algorithm to report all solutions to the turnpike reconstruction problem. Exploit additive constraints when possible to minimize the search. With proper pruning, problems with hundreds of points can be solved reliably.

Combinatorial Optimization

For each of the problems below, either (1) implement a combinatorial search program to solve it optimally for small instance, and/or (2) design and implement a simulated annealing heuristic to get reasonable solutions. How well does your program perform in practice?

7-7. *[5]* Design and implement an algorithm for solving the bandwidth minimization problem discussed in Section 13.2 (page 398).

7-8. *[5]* Design and implement an algorithm for solving the maximum satisfiability problem discussed in Section 14.10 (page 472).

7-9. *[5]* Design and implement an algorithm for solving the maximum clique problem discussed in Section 16.1 (page 525).

7-10. *[5]* Design and implement an algorithm for solving the minimum vertex coloring problem discussed in Section 16.7 (page 544).

7-11. *[5]* Design and implement an algorithm for solving the minimum edge coloring problem discussed in Section 16.8 (page 548).

7-12. *[5]* Design and implement an algorithm for solving the minimum feedback vertex set problem discussed in Section 16.11 (page 559).

7-13. *[5]* Design and implement an algorithm for solving the set cover problem discussed in Section 18.1 (page 621).

Interview Problems

7-14. *[4]* Write a function to find all permutations of the letters in a particular string.

7-15. *[4]* Implement an efficient algorithm for listing all k-element subsets of n items.

7-16. *[5]* An anagram is a rearrangement of the letters in a given string into a sequence of dictionary words, like *Steven Skiena* into *Vainest Knees*. Propose an algorithm to construct all the anagrams of a given string.

7-17. *[5]* Telephone keypads have letters on each numerical key. Write a program that generates all possible words resulting from translating a given digit sequence (e.g., 145345) into letters.

7-18. *[7]* You start with an empty room and a group of n people waiting outside. At each step, you may either admit one person into the room, or let one out. Can you arrange a sequence of 2^n steps, so that every possible combination of people is achieved exactly once?

7-19. *[4]* Use a random number generator (rng04) that generates numbers from $\{0, 1, 2, 3, 4\}$ with equal probability to write a random number generator that generates numbers from 0 to 7 (rng07) with equal probability. What are expected number of calls to rng04 per call of rng07?

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

7-1. "Little Bishops" – Programming Challenges 110801, UVA Judge 861.

7-2. "15-Puzzle Problem" – Programming Challenges 110802, UVA Judge 10181.

7-3. "Tug of War" – Programming Challenges 110805, UVA Judge 10032.

7-4. "Color Hash" – Programming Challenges 110807, UVA Judge 704.

8

Dynamic Programming

The most challenging algorithmic problems involve optimization, where we seek to find a solution that maximizes or minimizes some function. Traveling salesman is a classic optimization problem, where we seek the tour visiting all vertices of a graph at minimum total cost. But as shown in Chapter 1, it is easy to propose "algorithms" solving TSP that generate reasonable-looking solutions but did not *always* produce the minimum cost tour.

Algorithms for optimization problems require proof that they always return the best possible solution. Greedy algorithms that make the best local decision at each step are typically efficient but usually do not guarantee global optimality. Exhaustive search algorithms that try all possibilities and select the best always produce the optimum result, but usually at a prohibitive cost in terms of time complexity.

Dynamic programming combines the best of both worlds. It gives us a way to design custom algorithms that systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency). By storing the *consequences* of all possible decisions and using this information in a systematic way, the total amount of work is minimized.

Once you understand it, dynamic programming is probably the easiest algorithm design technique to apply in practice. In fact, I find that dynamic programming algorithms are often easier to reinvent than to try to look up in a book. That said, *until* you understand dynamic programming, it seems like magic. You must figure out the trick before you can use it.

Dynamic programming is a technique for efficiently implementing a recursive algorithm by storing partial results. The trick is seeing whether the naive recursive algorithm computes the same subproblems over and over and over again. If so, storing the answer for each subproblems in a table to look up instead of recompute

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_8,
© Springer-Verlag London Limited 2008

can lead to an efficient algorithm. Start with a recursive algorithm or definition. Only once we have a correct recursive algorithm do we worry about speeding it up by using a results matrix.

Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent *left to right* order among components. Left-to-right objects includes: character strings, rooted trees, polygons, and integer sequences. Dynamic programming is best learned by carefully studying examples until things start to click. We present three war stories where dynamic programming played the decisive role to demonstrate its utility in practice.

8.1 Caching vs. Computation

Dynamic programming is essentially a tradeoff of space for time. Repeatedly recomputing a given quantity is harmless unless the time spent doing so becomes a drag on performance. Then we are better off storing the results of the initial computation and looking them up instead of recomputing them again.

The tradeoff between space and time exploited in dynamic programming is best illustrated when evaluating recurrence relations such as the Fibonacci numbers. We look at three different programs for computing them below.

8.1.1 Fibonacci Numbers by Recursion

The Fibonacci numbers were originally defined by the Italian mathematician Fibonacci in the thirteenth century to model the growth of rabbit populations. Rabbits breed, well, like rabbits. Fibonacci surmised that the number of pairs of rabbits born in a given year is equal to the number of pairs of rabbits born in each of the two previous years, starting from one pair of rabbits in the first year. To count the number of rabbits born in the nth year, he defined the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

with basis cases $F_0 = 0$ and $F_1 = 1$. Thus, $F_2 = 1$, $F_3 = 2$, and the series continues $\{3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots\}$. As it turns out, Fibonacci's formula didn't do a very good job of counting rabbits, but it does have a host of interesting properties.

Since they are defined by a recursive formula, it is easy to write a recursive program to compute the nth Fibonacci number. A recursive function algorithm written in C looks like this:

```
long fib_r(int n)
{
        if (n == 0) return(0);
        if (n == 1) return(1);

        return(fib_r(n-1) + fib_r(n-2));
}
```

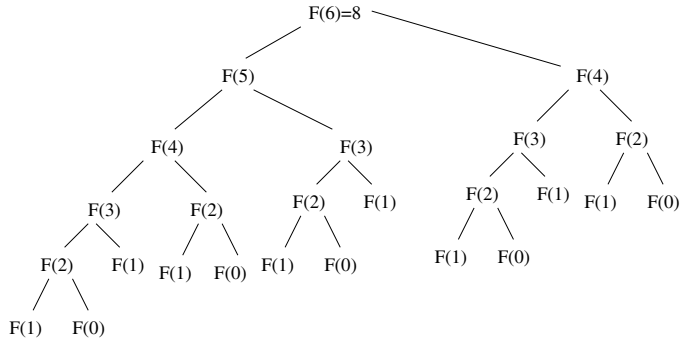

Figure 8.1: The computation tree for computing Fibonacci numbers recursively

The course of execution for this recursive algorithm is illustrated by its *recursion tree*, as illustrated in Figure 8.1. This tree is evaluated in a depth-first fashion, as are all recursive algorithms. I encourage you to trace this example by hand to refresh your knowledge of recursion.

Note that $F(4)$ is computed on both sides of the recursion tree, and $F(2)$ is computed no less than five times in this small example. The weight of all this redundancy becomes clear when you run the program. It took more than 7 minutes for my program to compute the first 45 Fibonacci numbers. You could probably do it faster by hand using the right algorithm.

How much time does this algorithm take to compute $F(n)$? Since $F_{n+1}/F_n \approx \phi = (1+\sqrt{5})/2 \approx 1.61803$, this means that $F_n > 1.6^n$. Since our recursion tree has only 0 and 1 as leaves, summing up to such a large number means we must have at least 1.6^n leaves or procedure calls! This humble little program takes exponential time to run!

8.1.2 Fibonacci Numbers by Caching

In fact, we can do much better. We can explicitly store (or *cache*) the results of each Fibonacci computation $F(k)$ in a table data structure indexed by the parameter k. The key to avoiding recomputation is to explicitly check for the value before trying to compute it:

```
#define MAXN    45       /* largest interesting n */
#define UNKNOWN   -1     /* contents denote an empty cell */
long f[MAXN+1];          /* array for caching computed fib values */
```

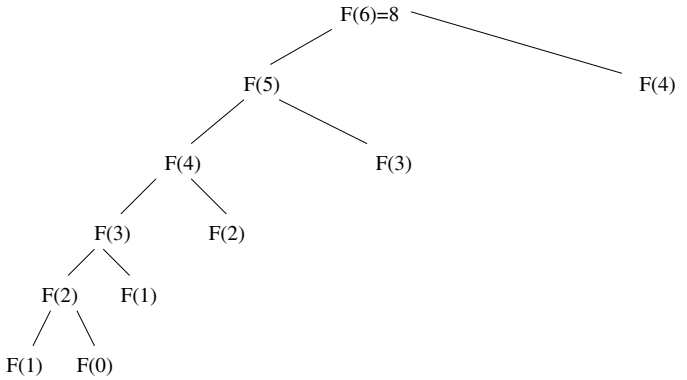

Figure 8.2: The Fibonacci computation tree when caching values

```
long fib_c(int n)
{
        if (f[n] == UNKNOWN)
                f[n] = fib_c(n-1) + fib_c(n-2);

        return(f[n]);
}


long fib_c_driver(int n)
{
        int i;                  /* counter */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = UNKNOWN;

        return(fib_c(n));
}
```

To compute $F(n)$, we call `fib_c_driver(n)`. This initializes our cache to the two values we initially know ($F(0)$ and $F(1)$) as well as the `UNKNOWN` flag for all the rest we don't. It then calls a look-before-crossing-the-street version of the recursive algorithm.

This cached version runs instantly up to the largest value that can fit in a long integer. The new recursion tree (Figure 8.2) explains why. There is no meaningful branching, because only the left-side calls do computation. The right-side calls find what they are looking for in the cache and immediately return.

What is the running time of this algorithm? The recursion tree provides more of a clue than the code. In fact, it computes $F(n)$ in linear time (in other words, $O(n)$ time) because the recursive function `fib_c(k)` is called exactly twice for each value $0 \leq k \leq n$.

This general method of explicitly caching results from recursive calls to avoid recomputation provides a simple way to get *most* of the benefits of full dynamic programming, so it is worth a more careful look. In principle, such caching can be employed on any recursive algorithm. However, storing partial results would have done absolutely no good for such recursive algorithms as *quicksort*, *backtracking*, and *depth-first search* because all the recursive calls made in these algorithms have distinct *parameter values*. It doesn't pay to store something you will never refer to again.

Caching makes sense only when the space of distinct parameter values is modest enough that we can afford the cost of storage. Since the argument to the recursive function `fib_c(k)` is an integer between 0 and n, there are only $O(n)$ values to cache. A linear amount of space for an exponential amount of time is an excellent tradeoff. But as we shall see, we can do even better by eliminating the recursion completely.

> *Take-Home Lesson:* Explicit caching of the results of recursive calls provides *most* of the benefits of dynamic programming, including usually the same running time as the more elegant full solution. If you prefer doing extra programming to more subtle thinking, you can stop here.

8.1.3 Fibonacci Numbers by Dynamic Programming

We can calculate F_n in linear time more easily by explicitly specifying the order of evaluation of the recurrence relation:

```
long fib_dp(int n)
{
        int i;              /* counter */
        long f[MAXN+1];     /* array to cache computed fib values */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = f[i-1]+f[i-2];

        return(f[n]);
}
```

We have removed all recursive calls! We evaluate the Fibonacci numbers from smallest to biggest and store all the results, so we know that we have F_{i-1} and F_{i-2} ready whenever we need to compute F_i. The linearity of this algorithm should

be apparent. Each of the n values is computed as the simple sum of two integers in total $O(n)$ time and space.

More careful study shows that we do not need to store all the intermediate values for the entire period of execution. Because the recurrence depends on two arguments, we only need to retain the last two values we have seen:

```
long fib_ultimate(int n)
{
        int i;                  /* counter */
        long back2=0, back1=1;  /* last two values of f[n] */
        long next;              /* placeholder for sum */

        if (n == 0) return (0);

        for (i=2; i<n; i++) {
                next = back1+back2;
                back2 = back1;
                back1 = next;
        }
        return(back1+back2);
}
```

This analysis reduces the storage demands to constant space with no asymptotic degradation in running time.

8.1.4 Binomial Coefficients

We now show how to compute the *binomial coefficients* as another illustration of how to eliminate recursion by specifying the order of evaluation. The binomial coefficients are the most important class of counting numbers, where $\binom{n}{k}$ counts the number of ways to choose k things out of n possibilities.

How do you compute the binomial coefficients? First, $\binom{n}{k} = n!/((n-k)!k!)$, so in principle you can compute them straight from factorials. However, this method has a serious drawback. Intermediate calculations can easily cause arithmetic overflow, even when the final coefficient fits comfortably within an integer.

A more stable way to compute binomial coefficients is using the recurrence relation implicit in the construction of Pascal's triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

m / n	0	1	2	3	4	5
0	A					
1	B	G				
2	C	1	H			
3	D	2	3	I		
4	E	4	5	6	J	
5	F	7	8	9	10	K

m / n	0	1	2	3	4	5
0	1					
1	1	1				
2	1	1	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	**5**	1

Figure 8.3: Evaluation order for `binomial_coefficient` at $M[5,4]$ (l). Initialization conditions A-K, recurrence evaluations 1-10. Matrix contents after evaluation (r)

Each number is the sum of the two numbers directly above it. The recurrence relation implicit in this is that

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Why does this work? Consider whether the nth element appears in one of the $\binom{n}{k}$ subsets of k elements. If so, we can complete the subset by picking $k-1$ other items from the other $n-1$. If not, we must pick all k items from the remaining $n-1$. There is no overlap between these cases, and all possibilities are included, so the sum counts all k subsets.

No recurrence is complete without basis cases. What binomial coefficient values do we know without computing them? The left term of the sum eventually drives us down to $\binom{n-k}{0}$. How many ways are there to choose 0 things from a set? Exactly one, the empty set. If this is not convincing, then it is equally good to accept that $\binom{m}{1} = m$ as the basis case. The right term of the sum runs us up to $\binom{k}{k}$. How many ways are there to choose k things from a k-element set? Exactly one—the complete set. Together, these basis cases and the recurrence define the binomial coefficients on all interesting values.

The best way to evaluate such a recurrence is to build a table of possible values up to the size that you are interested in:

Figure 8.3 demonstrates a proper evaluation order for the recurrence. The initialized cells are marked from A-K, denoting the order in which they were assigned values. Each remaining cell is assigned the sum of the cell directly above it and the cell immediately above and to the left. The triangle of cells marked 1 to 10 denote the evaluation order in computing $\binom{5}{4} = 5$ using the code below:

```
long binomial_coefficient(n,m)
int n,m;                          /* computer n choose m */
{
        int i,j;                  /* counters */
        long bc[MAXN][MAXN];      /* table of binomial coefficients */

        for (i=0; i<=n; i++) bc[i][0] = 1;

        for (j=0; j<=n; j++) bc[j][j] = 1;

        for (i=1; i<=n; i++)
                for (j=1; j<i; j++)
                        bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

        return( bc[n][m] );
}
```

Study this function carefully to see how we did it. The rest of this chapter will focus more on formulating and analyzing the appropriate recurrence than the mechanics of table manipulation demonstrated here.

8.2 Approximate String Matching

Searching for patterns in text strings is a problem of unquestionable importance. Section 3.7.2 (page 91) presented algorithms for *exact* string matching—finding where the pattern string P occurs as a substring of the text string T. Life is often not that simple. Words in either the text or pattern can be mispelled (sic), robbing us of exact similarity. Evolutionary changes in genomic sequences or language usage imply that we often search with archaic patterns in mind: "Thou shalt not kill" morphs over time into "You should not murder."

How can we search for the substring closest to a given pattern to compensate for spelling errors? To deal with inexact string matching, we must first define a cost function telling us how far apart two strings are—i.e., a distance measure between pairs of strings. A reasonable distance measure reflects the number of *changes* that must be made to convert one string to another. There are three natural types of changes:

- *Substitution* – Replace a single character from pattern P with a different character in text T, such as changing "shot" to "spot."
- *Insertion* – Insert a single character into pattern P to help it match text T, such as changing "ago" to "agog."
- *Deletion* – Delete a single character from pattern P to help it match text T, such as changing "hour" to "our."

Properly posing the question of string similarity requires us to set the cost of each of these string transform operations. Assigning each operation an equal cost of 1 defines the *edit distance* between two strings. Approximate string matching arises in many applications, as discussed in Section 18.4 (page 631).

Approximate string matching seems like a difficult problem, because we must decide exactly where to delete and insert (potentially) many characters in pattern and text. But let us think about the problem in reverse. What information would we like to have to make the final decision? What can happen to the last character in the matching for each string?

8.2.1 Edit Distance by Recursion

We can define a recursive algorithm using the observation that the last character in the string must either be matched, substituted, inserted, or deleted. Chopping off the characters involved in this last edit operation leaves a pair of smaller strings. Let i and j be the last character of the relevant prefix of P and T, respectively. There are three pairs of shorter strings after the last operation, corresponding to the strings after a match/substitution, insertion, or deletion. *If* we knew the cost of editing these three pairs of smaller strings, we could decide which option leads to the best solution and choose that option accordingly. We *can* learn this cost through the magic of recursion.

More precisely, let $D[i, j]$ be the minimum number of differences between $P_1, P_2, \ldots, P_i$ and the segment of T ending at j. $D[i, j]$ is the *minimum* of the three possible ways to extend smaller strings:

- If $(P_i = T_j)$, then $D[i-1, j-1]$, else $D[i-1, j-1]+1$. This means we either match or substitute the ith and jth characters, depending upon whether the tail characters are the same.

- $D[i-1, j]+1$. This means that there is an extra character in the pattern to account for, so we do not advance the text pointer and pay the cost of an insertion.

- $D[i, j-1]+1$. This means that there is an extra character in the text to remove, so we do not advance the pattern pointer and pay the cost of a deletion.

```
#define MATCH       0       /* enumerated type symbol for match */
#define INSERT      1       /* enumerated type symbol for insert */
#define DELETE      2       /* enumerated type symbol for delete */
```

```
int string_compare(char *s, char *t, int i, int j)
{
        int k;                  /* counter */
        int opt[3];             /* cost of the three options */
        int lowest_cost;        /* lowest cost */

        if (i == 0) return(j * indel(' '));
        if (j == 0) return(i * indel(' '));

        opt[MATCH] = string_compare(s,t,i-1,j-1) + match(s[i],t[j]);
        opt[INSERT] = string_compare(s,t,i,j-1) + indel(t[j]);
        opt[DELETE] = string_compare(s,t,i-1,j) + indel(s[i]);

        lowest_cost = opt[MATCH];
        for (k=INSERT; k<=DELETE; k++)
                if (opt[k] < lowest_cost) lowest_cost = opt[k];

        return( lowest_cost );
}
```

This program is absolutely correct—convince yourself. It also turns out to be impossibly slow. Running on my computer, the computation takes several seconds to compare two 11-character strings, and disappears into Never-Never Land on anything longer.

Why is the algorithm so slow? It takes exponential time because it recomputes values again and again and again. At every position in the string, the recursion branches three ways, meaning it grows at a rate of at least 3^n—indeed, even faster since most of the calls reduce only one of the two indices, not both of them.

8.2.2 Edit Distance by Dynamic Programming

So, how can we make this algorithm practical? The important observation is that most of these recursive calls are computing things that have been previously computed. How do we know? There can only be $|P| \cdot |T|$ possible unique recursive calls, since there are only that many distinct (i, j) pairs to serve as the argument parameters of recursive calls. By storing the values for each of these (i, j) pairs in a table, we just look them up as needed and avoid recomputing them.

A table-based, dynamic programming implementation of this algorithm is given below. The table is a two-dimensional matrix m where each of the $|P| \cdot |T|$ cells contains the cost of the optimal solution to a subproblem, as well as a parent pointer explaining how we got to this location:

```
typedef struct {
        int cost;               /* cost of reaching this cell */
        int parent;             /* parent cell */
} cell;

cell m[MAXLEN+1][MAXLEN+1];     /* dynamic programming table */
```

Our dynamic programming implementation has three differences from the recursive version. First, it gets its intermediate values using table lookup instead of recursive calls. Second, it updates the `parent` field of each cell, which will enable us to reconstruct the edit sequence later. Third, it is implemented using a more general `goal_cell()` function instead of just returning `m[|P|][|T|].cost`. This will enable us to apply this routine to a wider class of problems.

```
int string_compare(char *s, char *t)
{
        int i,j,k;              /* counters */
        int opt[3];             /* cost of the three options */

        for (i=0; i<MAXLEN; i++) {
                row_init(i);
                column_init(i);
        }

        for (i=1; i<strlen(s); i++) {
                for (j=1; j<strlen(t); j++) {
                        opt[MATCH] = m[i-1][j-1].cost + match(s[i],t[j]);
                        opt[INSERT] = m[i][j-1].cost + indel(t[j]);
                        opt[DELETE] = m[i-1][j].cost + indel(s[i]);

                        m[i][j].cost = opt[MATCH];
                        m[i][j].parent = MATCH;
                        for (k=INSERT; k<=DELETE; k++)
                                if (opt[k] < m[i][j].cost) {
                                        m[i][j].cost = opt[k];
                                        m[i][j].parent = k;
                                }
                }
        }
        goal_cell(s,t,&i,&j);
        return( m[i][j].cost );
}
```

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
:		**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	**1**	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	**2**	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	3	**2**	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	4	3	**2**	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	5	4	3	**2**	3	4	5	6	6	7	7	8	9	10
s:	6	6	6	5	4	3	**2**	3	4	5	6	7	8	8	9	10
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6	7	8	9	10
a:	8	8	8	7	6	5	4	3	3	**4**	5	6	7	8	9	10
l:	9	9	9	8	7	6	5	4	4	4	**4**	5	6	7	8	9
t:	10	10	10	9	8	7	6	5	5	5	5	**5**	6	7	8	8
-:	11	11	11	10	9	8	7	6	6	6	6	6	**5**	6	7	8
n:	12	12	12	11	10	9	8	7	7	7	7	7	6	**5**	6	7
o:	13	13	13	12	11	10	9	8	7	8	8	8	7	6	**5**	6
t:	14	14	14	13	12	11	10	9	8	8	9	9	8	7	6	**5**

Figure 8.4: Example of a dynamic programming matrix for editing distance computation, with the optimal alignment path highlighted in bold

Be aware that we adhere to somewhat unusual string and index conventions in the routine above. In particular, we assume that each string has been padded with an initial blank character, so the first real character of string `s` sits in `s[1]`. Why did we do this? It enables us to keep the matrix `m` indices in sync with those of the strings for clarity. Recall that we must dedicate the zeroth row and column of `m` to store the boundary values matching the empty prefix. Alternatively, we could have left the input strings intact and just adjusted the indices accordingly.

To determine the value of cell (i, j), we need three values sitting and waiting for us—namely, the cells $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$. Any evaluation order with this property will do, including the row-major order used in this program.[1]

As an example, we show the cost matrices for turning $p =$ "thou shalt not" into $t =$ "you should not" in five moves in Figure 8.4.

8.2.3 Reconstructing the Path

The string comparison function returns the cost of the optimal alignment, but not the alignment itself. Knowing you can convert "thou shalt not" to "you should not" in only five moves is dandy, but what is the sequence of editing operations that does it?

The possible solutions to a given dynamic programming problem are described by paths through the dynamic programming matrix, starting from the initial configuration (the pair of empty strings $(0, 0)$) down to the final goal state (the pair

[1]Suppose we create a graph with a vertex for every matrix cell, and a directed edge (x, y), when the value of cell x is needed to compute the value of cell y. Any topological sort on the resulting DAG (why must it be a DAG?) defines an acceptable evaluation order.

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	0	**-1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1
t:	1	**2**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h:	2	2	**0**	0	0	0	0	0	1	1	1	1	1	1	1	1
o:	3	2	0	**0**	0	0	0	0	0	1	1	1	1	1	0	1
u:	4	2	0	2	**0**	1	1	1	1	0	1	1	1	1	1	1
-:	5	2	0	2	2	**0**	1	1	1	1	0	0	0	1	1	1
s:	6	2	0	2	2	2	**0**	1	1	1	1	0	0	0	0	0
h:	7	2	0	2	2	2	2	**0**	**1**	1	1	1	1	1	0	0
a:	8	2	0	2	2	2	2	2	0	**0**	0	0	0	0	0	0
l:	9	2	0	2	2	2	2	2	0	0	**0**	1	1	1	1	1
t:	10	2	0	2	2	2	2	2	0	0	0	**0**	0	0	0	0
-:	11	2	0	2	2	0	2	2	0	0	0	0	**0**	1	1	1
n:	12	2	0	2	2	2	2	2	0	0	0	0	2	**0**	1	1
o:	13	2	0	0	2	2	2	2	0	0	0	0	2	2	**0**	1
t:	14	2	0	2	2	2	2	2	2	0	0	0	2	2	2	**0**

Figure 8.5: Parent matrix for edit distance computation, with the optimal alignment path highlighted in bold

of full strings ($|P|, |T|$)). The key to building the solution is to reconstruct the decisions made at every step along the optimal path that leads to the goal state. These decisions have been recorded in the `parent` field of each array cell.

Reconstructing these decisions is done by walking backward from the goal state, following the `parent` pointer back to an earlier cell. We repeat this process until we arrive back at the initial cell. The `parent` field for `m[i,j]` tells us whether the operation at (i, j) was `MATCH`, `INSERT`, or `DELETE`. Tracing back through the parent matrix in Figure 8.5 yields the edit sequence `DSMMMMMISMSMMMM` from "thou-shalt-not" to "you-should-not"—meaning delete the first "t", replace the "h" with "y", match the next five characters before inserting an "o", replace "a" with "u", and finally replace the "t" with a "d."

Walking backward reconstructs the solution in reverse order. However, clever use of recursion can do the reversing for us:

```
reconstruct_path(char *s, char *t, int i, int j)
{
        if (m[i][j].parent == -1) return;

        if (m[i][j].parent == MATCH) {
                reconstruct_path(s,t,i-1,j-1);
                match_out(s, t, i, j);
                return;
        }
        if (m[i][j].parent == INSERT) {
                reconstruct_path(s,t,i,j-1);
                insert_out(t,j);
```

```
                return;
        }
        if (m[i][j].parent == DELETE) {
                reconstruct_path(s,t,i-1,j);
                delete_out(s,i);
                return;
        }
}
```

For many problems, including edit distance, the tour can be reconstructed from the cost matrix without explicitly retaining the last-move pointer array. The trick is working backward from the costs of the three possible ancestor cells and corresponding string characters to reconstruct the move that took you to the current cell at the given cost.

8.2.4 Varieties of Edit Distance

The `string_compare` and path reconstruction routines reference several functions that we have not yet defined. These fall into four categories:

- *Table Initialization* – The functions `row_init` and `column_init` initialize the zeroth row and column of the dynamic programming table, respectively. For the string edit distance problem, cells $(i, 0)$ and $(0, i)$ correspond to matching length-i strings against the empty string. This requires exactly i insertions/deletions, so the definition of these functions is clear:

```
row_init(int i)
{
 m[0][i].cost = i;
 if (i>0)
     m[0][i].parent =  INSERT;
 else
     m[0][i].parent = -1;
}
```

```
column_init(int i)
{
 m[i][0].cost = i;
 if (i>0)
     m[i][0].parent = DELETE;
 else
     m[i][0].parent = -1;
}
```

- *Penalty Costs* – The functions `match(c,d)` and `indel(c)` present the costs for transforming character c to d and inserting/deleting character c. For standard edit distance, `match` should cost nothing if the characters are identical, and 1 otherwise; while `indel` returns 1 regardless of what the argument is. But application-specific cost functions can be employed, perhaps more forgiving of replacements located near each other on standard keyboard layouts or characters that sound or look similar.

```
int match(char c, char d)
{
  if (c == d) return(0);
  else return(1);
}
```

```
int indel(char c)
{
  return(1);
}
```

- *Goal Cell Identification* – The function `goal_cell` returns the indices of the cell marking the endpoint of the solution. For edit distance, this is defined by the length of the two input strings. However, other applications we will soon encounter do not have fixed goal locations.

```
goal_cell(char *s, char *t, int *i, int *j)
{
  *i = strlen(s) - 1;
  *j = strlen(t) - 1;
}
```

- *Traceback Actions* – The functions `match_out`, `insert_out`, and `delete_out` perform the appropriate actions for each edit operation during traceback. For edit distance, this might mean printing out the name of the operation or character involved, as determined by the needs of the application.

```
insert_out(char *t, int j)
{
        printf("I");
}

delete_out(char *s, int i)
{
        printf("D");
}
```

```
match_out(char *s, char *t,
          int i, int j)
{
    if (s[i]==t[j]) printf("M");
    else printf("S");
}
```

All of these functions are quite simple for edit distance computation. However, we must confess it is difficult to get the boundary conditions and index manipulations correctly. Although dynamic programming algorithms are easy to design once you understand the technique, getting the details right requires carefully thinking and thorough testing.

This may seem to be a lot of infrastructure to develop for such a simple algorithm. However, several important problems can now be solved as special cases of edit distance using only minor changes to some of these stub functions:

- *Substring Matching* – Suppose that we want to find where a short pattern P best occurs within a long text T—say, searching for "Skiena" in all its misspellings (Skienna, Skena, Skina, ...) within a long file. Plugging this

search into our original edit distance function will achieve little sensitivity, since the vast majority of any edit cost will consist of deleting that which is not "Skiena" from the body of the text. Indeed, matching any scattered $\cdots S \cdots k \cdots i \cdots e \cdots n \cdots a$ and deleting the rest yields an optimal solution.

We want an edit distance search where the cost of starting the match is independent of the position in the text, so that a match in the middle is not prejudiced against. Now the goal state is not necessarily at the end of both strings, but the cheapest place to match the entire pattern somewhere in the text. Modifying these two functions gives us the correct solution:

```
row_init(int i)
{
        m[0][i].cost = 0;       /* note change */
        m[0][i].parent = -1;    /* note change */
}

goal_cell(char *s, char *t, int *i, int *j)
{
        int k;                  /* counter */

        *i = strlen(s) - 1;
        *j = 0;
        for (k=1; k<strlen(t); k++)
                if (m[*i][k].cost < m[*i][*j].cost) *j = k;
}
```

- *Longest Common Subsequence* – Perhaps we are interested in finding the longest scattered string of characters included within both strings. Indeed, this problem will be discussed in Section 18.8. The *longest common subsequence* (LCS) between "democrat" and "republican" is *eca*.

  A common subsequence is defined by all the identical-character matches in an edit trace. To maximize the number of such matches, we must prevent substitution of nonidentical characters. With substitution forbidden, the only way to get rid of the noncommon subsequence is through insertion and deletion. The minimum cost alignment has the fewest such "in-dels", so it must preserve the longest common substring. We get the alignment we want by changing the match-cost function to make substitutions expensive:

```
int match(char c, char d)
{
        if (c == d) return(0);
        else return(MAXLEN);
}
```