"There isn't a whole lot of context when you type in three letters of stock market code."

"Sure, but there would be plenty of context if we typed in English sentences. I'll bet that we could reconstruct English text correctly if they were typed in a telephone at one keystroke per letter."

The guy from Periphonics gave me a disinterested look, then continued the tour. But when I got back to the office, I decided to give it a try.

Not all letters are equally likely to be typed on a telephone. In fact, not all letters *can* be typed, since Q and Z are not labeled on a standard American telephone. Therefore, we adopted the convention that Q, Z, and "space" all sat on the * key. We could take advantage of the uneven distribution of letter frequencies to help us decode the text. For example, if you hit the 3 key while typing English, you more likely meant to type an E than either a D or F. By taking into account the frequencies of a window of three characters (trigrams), we could predict the typed text. This is what happened when I tried it on the Gettysburg Address:

> enurraore ane reten yeasr ain our ectherr arotght eosti on ugis aootinent a oey oation aoncdivee in licesty ane eedicatee un uhe rrorosition uiat all oen are arectee e ual
>
> ony ye are enichde in a irect aitil yar uestini yhethes uiat oatioo or aoy oation ro aoncdivee ane ro eedicatee aan loni eneure ye are oet on a irect aattlediele oe uiat yar ye iate aone un eedicate a rostion oe uiat eiele ar a einal restini rlace eor uiore yin iere iate uhdis lives uiat uhe oation ogght live it is aluniethes eittini ane rrores uiat ye rioule en ugir
>
> att in a laries reore ye aan oou eedicate ye aan oou aoorearate ye aan oou ialloy ugis iroune the arate oen litini ane eeae yin rustgilee iere iate aoorearatee it ear aante our roor rowes un ade or eeuraat the yople yill little oote oor loni renences yiat ye ray iere att it aan oetes eosiet yiat uhfy eie iere it is eor ur uhe litini rathes un ae eedicatee iere un uhe undiniside yopl yhici uhfy yin entght iere iate uiur ear ro onaky aetancde it is rathes eor ur un ae iere eedicatee un uhe irect uarl rencinini adeore ur uiat eron uhere ioooree eeae ye uale inarearee eeuotion uo tiat aaure eor yhici uhfy iere iate uhe lart eull oearure oe eeuotioo tiat ye iere iggily rerolue uiat uhere eeae riall oou iate eide io

The trigram statistics did a decent job of translating it into Greek, but a terrible job of transcribing English. One reason was clear. This algorithm knew nothing about English words. If we coupled it with a dictionary, we might be onto something. But two words in the dictionary are often represented by the exact same string of phone codes. For an extreme example, the code string "22737" collides with eleven distinct English words, including *cases, cares, cards, capes, caper*, and *bases*. For our next attempt, we reported the unambiguous characters of any words that collided in the dictionary, and used trigrams to fill in the rest of the characters. We were rewarded with:

> eourscore and seven yearr ain our eatherr brought forth on this continent azoey nation conceivee in liberty and dedicatee uo uhe proposition that all men are createe equal
>
> ony ye are engagee in azipeat civil yar uestioi whether that nation or aoy nation ro conceivee and ro dedicatee aan long endure ye are oet on azipeat battlefield oe that yar ye iate aone uo dedicate a rostion oe that field ar a final perthni place for those yin here iate their lives that uhe nation oight live it is altogether fittinizane proper that ye should en this
>
> aut in a larges sense ye aan oou dedicate ye aan oou consecrate ye aan oou hallow this ground the arate men litioi and deae yin strugglee here iate consecratee it ear above our roor power uo ade or detract the world will little oote oor long remember what ye

> ray here aut it aan meter forget what uhfy die here it is for ur uhe litioi rather uo ae dedicatee here uo uhe toeioisgee york which uhfy yin fought here iate thus ear ro mocky advancee it is rather for ur uo ae here dedicatee uo uhe great task renagogoi adfore ur that from there honoree deae ye uale increasee devotion uo that aause for which uhfy here iate uhe last eull measure oe devotion that ye here highky resolve that there deae shall oou iate fide io vain that this nation under ioe shall iate azoey birth oe freedom and that ioternmenu oe uhe people ay uhe people for uhe people shall oou perish from uhe earth

If you were a student of American history, maybe you could recognize it, but you certainly couldn't read it. Somehow, we had to distinguish between the different dictionary words that got hashed to the same code. We could factor in the relative popularity of each word, but this still made too many mistakes.

At this point, I started working with Harald Rau on the project, who proved to be a great collaborator. First, he was a bright and persistent graduate student. Second, as a native German speaker, he believed every lie I told him about English grammar.

Harald built up a phone code reconstruction program along the lines of Figure 6.9. It worked on the input one sentence at a time, identifying dictionary words that matched each code string. The key problem was how to incorporate grammatical constraints.

"We can get good word-use frequencies and grammatical information from a big text database called the Brown Corpus. It contains thousands of typical English sentences, each parsed according to parts of speech. But how do we factor it all in?" Harald asked.

"Let's think about it as a graph problem," I suggested.

"*Graph problem?* What graph problem? Where is there even a graph?"

"Think of a sentence as a series of phone tokens, each representing a word in the sentence. Each phone token has a list of words from the dictionary that match it. How can we choose which one is right? Each possible sentence interpretation can be thought of as a path in a graph. The vertices of this graph are the complete set of possible word choices. There will be an edge from each possible choice for the ith word to each possible choice for the $(i+1)$st word. The cheapest path across this graph defines the best interpretation of the sentence."

"But all the paths look the same. They have the same number of edges. Wait. Now I see! We have to add weight to the edges to make the paths different."

"Exactly! The cost of an edge will reflect how likely it is that we will travel through the given pair of words. Perhaps we can count how often that pair of words occurred together in previous texts. Or we can weigh them by the part of speech of each word. Maybe nouns don't like to be next to nouns as much as they like being next to verbs."

"It will be hard to keep track of word-pair statistics, since there are so many of them. But we certainly know the frequency of each word. How can we factor that into things?"

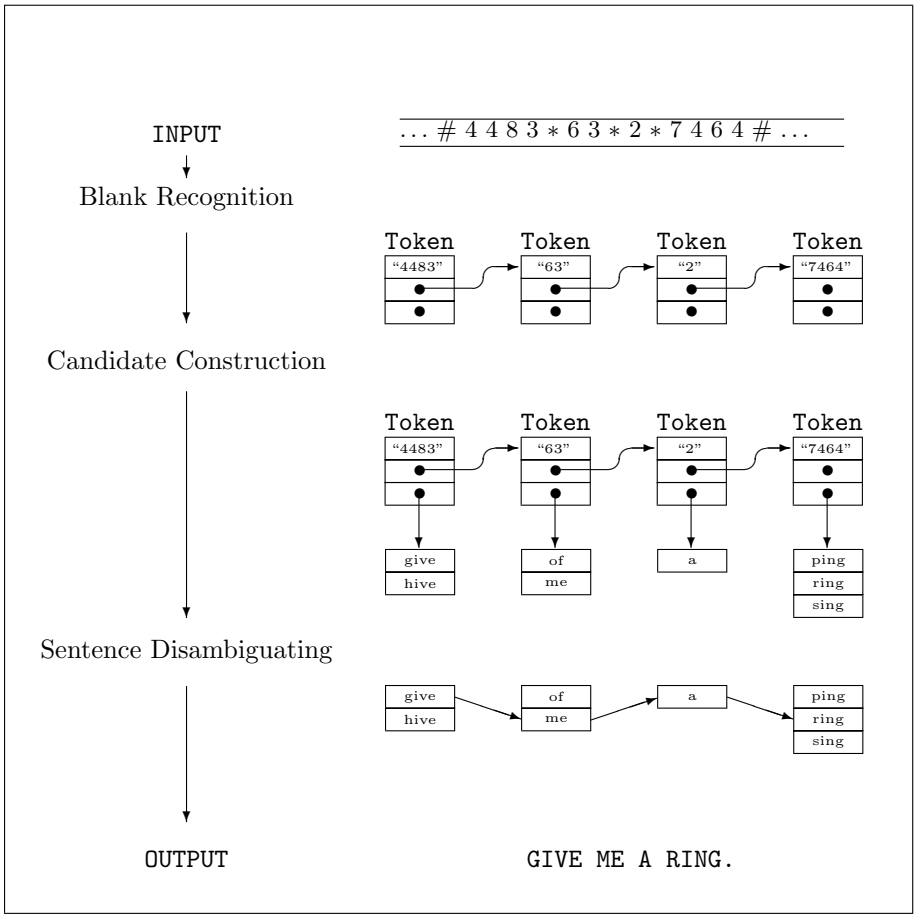

Figure 6.9: The phases of the telephone code reconstruction process

"We can pay a cost for walking through a particular vertex that depends upon the frequency of the word. Our best sentence will be given by the shortest path across the graph."

"But how do we figure out the relative weights of these factors?"

"First try what seems natural to you and then we can experiment with it."

Harald incorporated this shortest-path algorithm. With proper grammatical and statistical constraints, the system performed great. Look at the Gettysburg Address now, with all the reconstruction errors highlighted:

FOURSCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH ON THIS CONTINENT A NEW NATION CONCEIVED IN LIBERTY AND DEDICATED TO THE PROPOSITION THAT ALL MEN ARE CREATED EQUAL. NOW WE ARE

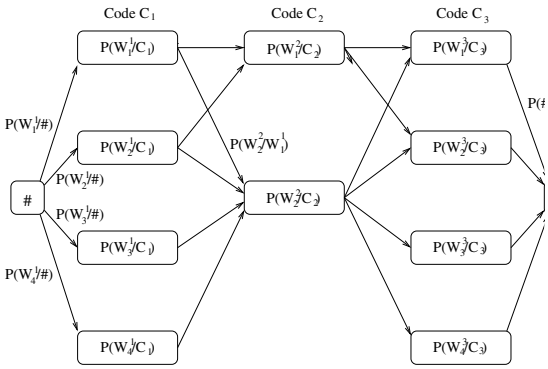

Figure 6.10: The minimum-cost path defines the best interpretation for a sentence

Text	characters	characters correct	non-blanks correct	words correct	time per character
Clinton Speeches	1,073,593	99.04%	98.86%	97.67%	0.97ms
Herland	278,670	98.24%	97.89%	97.02%	0.97ms
Moby Dick	1,123,581	96.85%	96.25%	94.75%	1.14ms
Bible	3,961,684	96.20%	95.39%	95.39%	1.33ms
Shakespeare	4,558,202	95.20%	94.21%	92.86%	0.99ms

Figure 6.11: Telephone-code reconstruction applied to several text samples

ENGAGED IN A GREAT CIVIL WAR TESTING WHETHER THAT NATION OR ANY NATION SO CONCEIVED AND SO DEDICATED CAN LONG ENDURE. WE ARE MET ON A GREAT BATTLEFIELD OF THAT **WAS**. WE HAVE COME TO DEDICATE A PORTION OF THAT FIELD AS A FINAL **SERVING** PLACE FOR THOSE WHO HERE **HAVE** THEIR LIVES THAT THE NATION MIGHT LIVE. IT IS ALTOGETHER FITTING AND PROPER THAT WE SHOULD DO THIS. BUT IN A LARGER SENSE WE CAN NOT DEDICATE WE CAN NOT CONSECRATE WE CAN NOT HALLOW THIS GROUND. THE BRAVE MEN LIVING AND DEAD WHO STRUGGLED HERE HAVE CONSECRATED IT FAR ABOVE OUR POOR POWER TO ADD OR DETRACT. THE WORLD WILL LITTLE NOTE NOR LONG REMEMBER WHAT WE SAY HERE BUT IT CAN NEVER FORGET WHAT THEY DID HERE. IT IS FOR US THE LIVING RATHER TO BE DEDICATED HERE TO THE UNFINISHED WORK WHICH THEY WHO FOUGHT HERE HAVE THUS FAR SO NOBLY ADVANCED. IT IS RATHER FOR US TO BE HERE DEDICATED TO THE GREAT TASK REMAINING BEFORE US THAT FROM THESE HONORED DEAD WE TAKE INCREASED DEVOTION TO THAT CAUSE FOR WHICH THEY HERE **HAVE** THE LAST FULL MEASURE OF DEVOTION THAT WE HERE HIGHLY RESOLVE THAT THESE DEAD SHALL NOT HAVE DIED IN VAIN THAT THIS NATION UNDER GOD SHALL HAVE A NEW BIRTH OF FREEDOM AND THAT GOVERNMENT OF THE PEOPLE BY THE PEOPLE FOR THE PEOPLE SHALL NOT PERISH FROM THE EARTH.

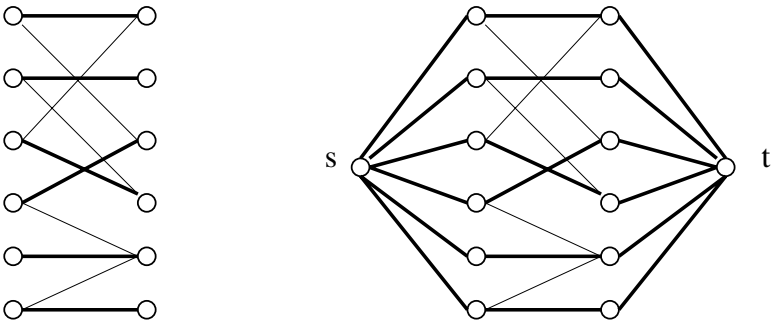

Figure 6.12: Bipartite graph with a maximum matching highlighted (on left). The corresponding network flow instance highlighting the maximum $s-t$ flow (on right).

While we still made a few mistakes, the results are clearly good enough for many applications. Periphonics certainly thought so, for they licensed our program to incorporate into their products. Figure 6.11 shows that we were able to reconstruct correctly over 99% of the characters in a megabyte of President Clinton's speeches, so if Bill had phoned them in, we would certainly be able to understand what he was saying. The reconstruction time is fast enough, indeed faster than you can type it in on the phone keypad.

The constraints for many pattern recognition problems can be naturally formulated as shortest path problems in graphs. In fact, there is a particularly convenient dynamic programming solution for these problems (the Viterbi algorithm) that is widely used in speech and handwriting recognition systems. Despite the fancy name, the Viterbi algorithm is basically solving a shortest path problem on a DAG. Hunting for a graph formulation to solve any given problem is often a good idea.

6.5 Network Flows and Bipartite Matching

Edge-weighted graphs can be interpreted as a network of pipes, where the weight of edge (i, j) determines the *capacity* of the pipe. Capacities can be thought of as a function of the cross-sectional area of the pipe. A wide pipe might be able to carry 10 units of flow in a given time, where as a narrower pipe might only carry 5 units. The *network flow problem* asks for the maximum amount of flow which can be sent from vertices s to t in a given weighted graph G while respecting the maximum capacities of each pipe.

6.5.1 Bipartite Matching

While the network flow problem is of independent interest, its primary importance is in to solving other important graph problems. A classic example is bipartite matching. A *matching* in a graph $G = (V, E)$ is a subset of edges $E' \subset E$ such that no two edges of E' share a vertex. A matching pairs off certain vertices such that every vertex is in, at most, one such pair, as shown in Figure 6.12.

Graph G is *bipartite* or *two-colorable* if the vertices can be divided into two sets, L and R, such that all edges in G have one vertex in L and one vertex in R. Many naturally defined graphs are bipartite. For example, certain vertices may represent jobs to be done and the remaining vertices represent people who can potentially do them. The existence of edge (j, p) means that job j can be done by person p. Or let certain vertices represent boys and certain vertices represent girls, with edges representing compatible pairs. Matchings in these graphs have natural interpretations as job assignments or as marriages, and are the focus of Section 15.6 (page 498).

The largest bipartite matching can be readily found using network flow. Create a *source* node s that is connected to every vertex in L by an edge of weight 1. Create a *sink* node t and connect it to every vertex in R by an edge of weight 1. Finally, assign each edge in the bipartite graph G a weight of 1. Now, the maximum possible flow from s to t defines the largest matching in G. Certainly we can find a flow as large as the matching by using only the matching edges and their source-to-sink connections. Further, there can be no greater possible flow. How can we ever hope to get more than one flow unit through any vertex?

6.5.2 Computing Network Flows

Traditional network flow algorithms are based on the idea of *augmenting paths*, and repeatedly finding a path of positive capacity from s to t and adding it to the flow. It can be shown that the flow through a network is optimal if and only if it contains no augmenting path. Since each augmentation adds to the flow, we must eventually find the global maximum.

The key structure is the *residual flow graph*, denoted as $R(G, f)$, where G is the input graph and f is the current flow through G. This directed, edge-weighted $R(G, f)$ contains the same vertices as G. For each edge (i, j) in G with capacity $c(i, j)$ and flow $f(i, j)$, $R(G, f)$ may contain two edges:

(i) an edge (i, j) with weight $c(i, j) - f(i, j)$, if $c(i, j) - f(i, j) > 0$ and

(ii) an edge (j, i) with weight $f(i, j)$, if $f(i, j) > 0$.

The presence of edge (i, j) in the residual graph indicates that positive flow can be pushed from i to j. The weight of the edge gives the exact amount that can be pushed. A path in the residual flow graph from s to t implies that more flow can be pushed from s to t and the minimum edge weight on this path defines the amount of extra flow that can be pushed.

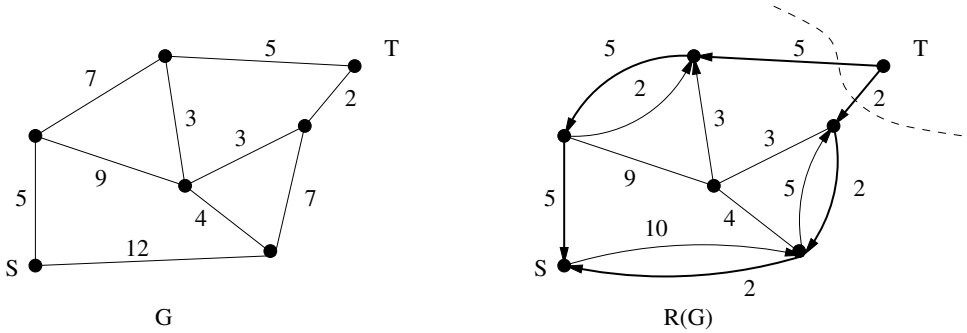

Figure 6.13: Maximum $s-t$ flow in a graph G (on left) showing the associated residual graph $R(G)$ and minimum $s-t$ cut (dotted line near t)

Figure 6.13 illustrates this idea. The maximum $s-t$ flow in graph G is 7. Such a flow is revealed by the two directed t to s paths in the residual graph $R(G)$ of capacities $2+5$, respectively. These flows completely saturate the capacity of the two edges incident to vertex t, so no augmenting path remains. Thus the flow is optimal. A set of edges whose deletion separates s from t (like the two edges incident to t) is called an s-t cut. Clearly, no s to t flow can exceed the weight of the minimum such cut. In fact, a flow equal to the minimum cut is always possible.

Take-Home Lesson: The maximum flow from s to t always equals the weight of the minimum s-t cut. Thus, flow algorithms can be used to solve general edge and vertex connectivity problems in graphs.

Implementation

We cannot do full justice to the theory of network flows here. However, it is instructive to see how augmenting paths can be identified and the optimal flow computed.

For each edge in the residual flow graph, we must keep track of both the amount of flow currently going through the edge, as well as its remaining *residual* capacity. Thus, we must modify our `edge` structure to accommodate the extra fields:

```
typedef struct {
        int v;                      /* neighboring vertex */
        int capacity;               /* capacity of edge */
        int flow;                   /* flow through edge */
        int residual;               /* residual capacity of edge */
        struct edgenode *next;      /* next edge in list */
} edgenode;
```

We use a breadth-first search to look for any path from source to sink that increases the total flow, and use it to augment the total. We terminate with the optimal flow when no such *augmenting* path exists.

```
netflow(flow_graph *g, int source, int sink)
{
      int volume;          /* weight of the augmenting path */

      add_residual_edges(g);

      initialize_search(g);
      bfs(g,source);

      volume = path_volume(g, source, sink, parent);

      while (volume > 0) {
              augment_path(g,source,sink,parent,volume);
              initialize_search(g);
              bfs(g,source);
              volume = path_volume(g, source, sink, parent);
      }
}
```

Any augmenting path from source to sink increases the flow, so we can use `bfs` to find such a path in the appropriate graph. We only consider network edges that have remaining capacity or, in other words, positive residual flow. The predicate below helps `bfs` distinguish between saturated and unsaturated edges:

```
bool valid_edge(edgenode *e)
{
        if (e->residual > 0) return (TRUE);
        else return(FALSE);
}
```

Augmenting a path transfers the maximum possible volume from the residual capacity into positive flow. This amount is limited by the path-edge with the smallest amount of residual capacity, just as the rate at which traffic can flow is limited by the most congested point.

```
int path_volume(flow_graph *g, int start, int end, int parents[])
{
     edgenode *e;                    /* edge in question */
     edgenode *find_edge();

     if (parents[end] == -1) return(0);

     e = find_edge(g,parents[end],end);

     if (start == parents[end])
             return(e->residual);
     else
             return( min(path_volume(g,start,parents[end],parents),
                         e->residual) );
}


edgenode *find_edge(flow_graph *g, int x, int y)
{
        edgenode *p;                    /* temporary pointer */

        p = g->edges[x];

        while (p != NULL) {
                if (p->v == y) return(p);
                p = p->next;
        }

        return(NULL);
}
```

Sending an additional unit of flow along directed edge (i, j) reduces the residual capacity of edge (i, j) but *increases* the residual capacity of edge (j, i). Thus, the act of augmenting a path requires modifying both forward and reverse edges for each link on the path.

```
augment_path(flow_graph *g, int start, int end, int parents[], int volume)
{
        edgenode *e;                    /* edge in question */
        edgenode *find_edge();

        if (start == end) return;

        e = find_edge(g,parents[end],end);
        e->flow += volume;
        e->residual -= volume;

        e = find_edge(g,end,parents[end]);
        e->residual += volume;

        augment_path(g,start,parents[end],parents,volume);
}
```

Initializing the flow graph requires creating directed flow edges (i, j) and (j, i) for each network edge $e = (i, j)$. Initial flows are all set to 0. The initial residual flow of (i, j) is set to the capacity of e, while the initial residual flow of (j, i) is set to 0.

Analysis

The augmenting path algorithm above eventually converges on the the optimal solution. However, each augmenting path may add only a little to the total flow, so, in principle, the algorithm might take an arbitrarily long time to converge.

However, Edmonds and Karp [EK72] proved that always selecting a *shortest* unweighted augmenting path guarantees that $O(n^3)$ augmentations suffice for optimization. In fact, the Edmonds-Karp algorithm is what is implemented above, since a breadth-first search from the source is used to find the next augmenting path.

6.6 Design Graphs, Not Algorithms

Proper modeling is the key to making effective use of graph algorithms. We have defined several graph properties, and developed algorithms for computing them. All told, about two dozen different graph problems are presented in the catalog, mostly in Sections 15 and 16. These classical graph problems provide a framework for modeling most applications.

The secret is learning to design graphs, not algorithms. We have already seen a few instances of this idea:

- The *maximum* spanning tree can be found by negating the edge weights of the input graph G and using a *minimum* spanning tree algorithm on the result. The most negative weight spanning tree will define the maximum weight tree in G.

- To solve bipartite matching, we constructed a special network flow graph such that the maximum flow corresponds to a maximum cardinality matching.

The applications below demonstrate the power of proper modeling. Each arose in a real-world application, and each can be modeled as a graph problem. Some of the modelings are quite clever, but they illustrate the versatility of graphs in representing relationships. As you read a problem, try to devise an appropriate graph representation before peeking to see how we did it.

Stop and Think: The Pink Panther's Passport to Peril

Problem: "I'm looking for an algorithm to design natural routes for video-game characters to follow through an obstacle-filled room. How should I do it?"

Solution: Presumably the desired route should look like a path that an intelligent being would choose. Since intelligent beings are either lazy or efficient, this should be modeled as a shortest path problem.

But what is the graph? One approach might be to lay a grid of points in the room. Create a vertex for each grid point that is a valid place for the character to stand; i.e. , that does not lie within an obstacle. There will be an edge between any pair of nearby vertices, weighted proportionally to the distance between them. Although direct geometric methods are known for shortest paths (see Section 15.4 (page 489)), it is easier to model this discretely as a graph. ∎

Stop and Think: Ordering the Sequence

Problem: "A DNA sequencing project generates experimental data consisting of small fragments. For each given fragment f, we know certain other fragments are forced to lie to the left of f, and certain other fragments are forced to be to the right of f. How can we find a consistent ordering of the fragments from left to right that satisfies all the constraints?"

Solution: Create a directed graph, where each fragment is assigned a unique vertex. Insert a directed edge $(\overrightarrow{l, f})$ from any fragment l that is forced to be to the left

of f, and a directed edge $(\overrightarrow{f, r})$ to any fragment r forced to be to the right of f. We seek an ordering of the vertices such that all the edges go from left to right. This is a *topological sort* of the resulting directed acyclic graph. The graph must be acyclic, because cycles make finding a consistent ordering impossible. ∎

Stop and Think: Bucketing Rectangles

Problem: "In my graphics work I need to solve the following problem. Given an arbitrary set of rectangles in the plane, how can I distribute them into a minimum number of buckets such that no subset of rectangles in any given bucket intersects another? In other words, there can not be any overlapping area between two rectangles in the same bucket."

Solution: We formulate a graph where each vertex is a rectangle, and there is an edge if two rectangles intersect. Each bucket corresponds to an *independent set* of rectangles, so there is no overlap between any two. A *vertex coloring* of a graph is a partition of the vertices into independent sets, so minimizing the number of colors is exactly what you want. ∎

Stop and Think: Names in Collision

Problem: "In porting code from UNIX to DOS, I have to shorten several hundred file names down to at most 8 characters each. I can't just use the first eight characters from each name, because "filename1" and "filename2" would be assigned the exact same name. How can I meaningfully shorten the names while ensuring that they do not collide?"

Solution: Construct a bipartite graph with vertices corresponding to each original file name f_i for $1 \leq i \leq n$, as well as a collection of acceptable shortenings for each name $f_{i1}, \ldots, f_{ik}$. Add an edge between each original and shortened name. We now seek a set of n edges that have no vertices in common, so each file name is mapped to a distinct acceptable substitute. *Bipartite matching*, discussed in Section 15.6 (page 498), is exactly this problem of finding an independent set of edges in a graph. ∎

Stop and Think: Separate the Text

Problem: "We need a way to separate the lines of text in the optical character-recognition system that we are building. Although there is some white space between the lines, problems like noise and the tilt of the page makes it hard to find. How can we do line segmentation?

Solution: Consider the following graph formulation. Treat each pixel in the image as a vertex in the graph, with an edge between two neighboring pixels. The weight of this edge should be proportional to how dark the pixels are. A segmentation between two lines is a path in this graph from the left to right side of the page. We seek a relatively straight path that avoids as much blackness as possible. This suggests that the *shortest path* in the pixel graph will likely find a good line segmentation. ∎

Take-Home Lesson: Designing novel graph algorithms is very hard, so don't do it. Instead, try to design graphs that enable you to use classical algorithms to model your problem.

Chapter Notes

Network flows are an advanced algorithmic technique, and recognizing whether a particular problem can be solved by network flow requires experience. We point the reader to books by Cook and Cunningham [CC97] and Ahuja, Magnanti, and Orlin [AMO93] for more detailed treatments of the subject.

The augmenting path method for network flows is due to Ford and Fulkerson [FF62]. Edmonds and Karp [EK72] proved that always selecting a *shortest* geodesic augmenting path guarantees that $O(n^3)$ augmentations suffice for optimization.

The phone code reconstruction system that was the subject of the war story is described in more technical detail in [RS96].

6.7 Exercises

Simulating Graph Algorithms

6-1. *[3]* For the graphs in Problem 5-1:

(a) Draw the spanning forest after every iteration of the main loop in Kruskal's algorithm.

(b) Draw the spanning forest after every iteration of the main loop in Prim's algorithm.

(c) Find the shortest path spanning tree rooted in A.

(d) Compute the maximum flow from A to H.

Minimum Spanning Trees

6-2. *[3]* Is the path between two vertices in a minimum spanning tree necessarily a shortest path between the two vertices in the full graph? Give a proof or a counterexample.

6-3. *[3]* Assume that all edges in the graph have distinct edge weights (i.e. , no pair of edges have the same weight). Is the path between a pair of vertices in a minimum spanning tree necessarily a shortest path between the two vertices in the full graph? Give a proof or a counterexample.

6-4. *[3]* Can Prim's and Kruskal's algorithm yield different minimum spanning trees? Explain why or why not.

6-5. *[3]* Does either Prim's and Kruskal's algorithm work if there are negative edge weights? Explain why or why not.

6-6. *[5]* Suppose we are *given* the minimum spanning tree T of a given graph G (with n vertices and m edges) and a new edge $e = (u, v)$ of weight w that we will add to G. Give an efficient algorithm to find the minimum spanning tree of the graph $G + e$. Your algorithm should run in $O(n)$ time to receive full credit.

6-7. *[5]* (a) Let T be a minimum spanning tree of a weighted graph G. Construct a new graph G' by adding a weight of k to every edge of G. Do the edges of T form a minimum spanning tree of G'? Prove the statement or give a counterexample.

(b) Let $P = \{s, \ldots, t\}$ describe a shortest weighted path between vertices s and t of a weighted graph G. Construct a new graph G' by adding a weight of k to every edge of G. Does P describe a shortest path from s to t in G'? Prove the statement or give a counterexample.

6-8. *[5]* Devise and analyze an algorithm that takes a weighted graph G and finds the smallest change in the cost to a non-MST edge that would cause a change in the minimum spanning tree of G. Your algorithm must be correct and run in polynomial time.

6-9. *[4]* Consider the problem of finding a minimum weight connected subset T of edges from a weighted connected graph G. The weight of T is the sum of all the edge weights in T.

(a) Why is this problem not just the minimum spanning tree problem? Hint: think negative weight edges.

(b) Give an efficient algorithm to compute the minimum weight connected subset T.

6-10. *[4]* Let $G = (V, E)$ be an undirected graph. A set $F \subseteq E$ of edges is called a *feedback-edge set* if every cycle of G has at least one edge in F.

(a) Suppose that G is unweighted. Design an efficient algorithm to find a minimum-size feedback-edge set.

(b) Suppose that G is a weighted undirected graph with positive edge weights. Design an efficient algorithm to find a minimum-weight feedback-edge set.

6-11. *[5]* Modify Prim's algorithm so that it runs in time $O(n \log k)$ on a graph that has only k different edges costs.

Union-Find

6-12. *[5]* Devise an efficient data structure to handle the following operations on a weighted directed graph:

(a) Merge two given components.

(b) Locate which component contains a given vertex v.

(c) Retrieve a minimum edge from a given component.

6-13. *[5]* Design a data structure that can perform a sequence of, m *union* and *find* operations on a universal set of n elements, consisting of a sequence of all *unions* followed by a sequence of all *finds*, in time $O(m + n)$.

Shortest Paths

6-14. *[3]* The *single-destination shortest path* problem for a directed graph seeks the shortest path *from* every vertex to a specified vertex v. Give an efficient algorithm to solve the single-destination shortest paths problem.

6-15. *[3]* Let $G = (V, E)$ be an undirected weighted graph, and let T be the shortest-path spanning tree rooted at a vertex v. Suppose now that all the edge weights in G are increased by a constant number k. Is T still the shortest-path spanning tree from v?

6-16. *[3]* Answer all of the following:

(a) Give an example of a weighted connected graph $G = (V, E)$ and a vertex v, such that the minimum spanning tree of G is the same as the shortest-path spanning tree rooted at v.

(b) Give an example of a weighted connected directed graph $G = (V, E)$ and a vertex v, such that the minimum-cost spanning tree of G is very different from the shortest-path spanning tree rooted at v.

(c) Can the two trees be completely disjointed?

6-17. *[3]* Either prove the following or give a counterexample:

(a) Is the path between a pair of vertices in a minimum spanning tree of an undirected graph necessarily the shortest (minimum weight) path?

(b) Suppose that the minimum spanning tree of the graph is unique. Is the path between a pair of vertices in a minimum spanning tree of an undirected graph necessarily the shortest (minimum weight) path?

6-18. *[5]* In certain graph problems, vertices have can have weights instead of or in addition to the weights of edges. Let C_v be the cost of vertex v, and $C_{(x,y)}$ the cost of the edge (x, y). This problem is concerned with finding the cheapest path between vertices a and b in a graph G. The cost of a path is the sum of the costs of the edges and vertices encountered on the path.

(a) Suppose that each edge in the graph has a weight of zero (while non-edges have a cost of ∞). Assume that $C_v = 1$ for all vertices $1 \leq v \leq n$ (i.e. , all vertices have the same cost). Give an *efficient* algorithm to find the cheapest path from a to b and its time complexity.

(b) Now suppose that the vertex costs are not constant (but are all positive) and the edge costs remain as above. Give an *efficient* algorithm to find the cheapest path from a to b and its time complexity.

(c) Now suppose that both the edge and vertex costs are not constant (but are all positive). Give an *efficient* algorithm to find the cheapest path from a to b and its time complexity.

6-19. *[5]* Let G be a weighted *directed* graph with n vertices and m edges, where all edges have positive weight. A directed cycle is a directed path that starts and ends at the same vertex and contains at least one edge. Give an $O(n^3)$ algorithm to find a directed cycle in G of minimum total weight. Partial credit will be given for an $O(n^2 m)$ algorithm.

6-20. *[5]* Can we modify Dijkstra's algorithm to solve the single-source *longest* path problem by changing *minimum* to *maximum*? If so, then prove your algorithm correct. If not, then provide a counterexample.

6-21. *[5]* Let $G = (V, E)$ be a weighted acyclic directed graph with possibly negative edge weights. Design a linear-time algorithm to solve the single-source shortest-path problem from a given source v.

6-22. *[5]* Let $G = (V, E)$ be a directed weighted graph such that all the weights are positive. Let v and w be two vertices in G and $k \leq |V|$ be an integer. Design an algorithm to find the shortest path from v to w that contains exactly k edges. Note that the path need not be simple.

6-23. *[5]* *Arbitrage* is the use of discrepancies in currency-exchange rates to make a profit. For example, there may be a small window of time during which 1 U.S. dollar buys 0.75 British pounds, 1 British pound buys 2 Australian dollars, and 1 Australian dollar buys 0.70 U.S. dollars. At such a time, a smart trader can trade one U.S. dollar and end up with $0.75 \times 2 \times 0.7 = 1.05$ U.S. dollars—a profit of 5%. Suppose that there are n currencies $c_1, ..., c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j. Devise and analyze an algorithm to determine the maximum value of

$$R[c_1, c_{i1}] \cdot R[c_{i1}, c_{i2}] \cdots R[c_{ik-1}, c_{ik}] \cdot R[c_{ik}, c_1]$$

Hint: think all-pairs shortest path.

Network Flow and Matching

6-24. *[3]* A matching in a graph is a set of disjoint edges—i.e., edges that do not share any vertices in common. Give a linear-time algorithm to find a maximum matching in a tree.

6-25. *[5]* An *edge cover* of an undirected graph $G = (V, E)$ is a set of edges such that each vertex in the graph is incident to at least one edge from the set. Give an efficient algorithm, based on matching, to find the minimum-size edge cover for G.

Programming Challenges

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

6-1. "Freckles" – Programming Challenges 111001, UVA Judge 10034.

6-2. "Necklace" – Programming Challenges 111002, UVA Judge 10054.

6-3. "Railroads" – Programming Challenges 111004, UVA Judge 10039.

6-4. "Tourist Guide" – Programming Challenges 111006, UVA Judge 10199.

6-5. "The Grand Dinner" – Programming Challenges 111007, UVA Judge 10249.

7

Combinatorial Search and Heuristic Methods

We can solve many problems to optimality using exhaustive search techniques, although the time complexity can be enormous. For certain applications, it may pay to spend extra time to be certain of the optimal solution. A good example occurs in testing a circuit or a program on all possible inputs. You can prove the correctness of the device by trying all possible inputs and verifying that they give the correct answer. Verifying correctness is a property to be proud of. However, claiming that it works correctly on all the inputs you tried is worth much less.

Modern computers have clock rates of a few *gigahertz*, meaning billions of operations per second. Since doing something interesting takes a few hundred instructions, you can hope to search millions of items per second on contemporary machines.

It is important to realize how big (or how small) one million is. One million permutations means all arrangements of roughly 10 or 11 objects, but not more. One million subsets means all combinations of roughly 20 items, but not more. Solving significantly larger problems requires carefully pruning the search space to ensure we look at only the elements that really matter.

In this section, we introduce backtracking as a technique for listing all possible solutions for a combinatorial algorithm problem. We illustrate the power of clever pruning techniques to speed up real search applications. For problems that are too large to contemplate using brute-force combinatorial search, we introduce heuristic methods such as simulated annealing. Such heuristic methods are important weapons in any practical algorist's arsenal.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_7,

7.1 Backtracking

Backtracking is a systematic way to iterate through all the possible configurations of a search space. These configurations may represent all possible arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). Other situations may demand enumerating all spanning trees of a graph, all paths between two vertices, or all possible ways to partition vertices into color classes.

What these problems have in common is that we must generate each one possible configuration exactly once. Avoiding both repetitions and missing configurations means that we must define a systematic generation order. We will model our combinatorial search solution as a vector $a = (a_1, a_2, ..., a_n)$, where each element a_i is selected from a finite ordered set S_i. Such a vector might represent an arrangement where a_i contains the ith element of the permutation. Or, the vector might represent a given subset S, where a_i is true if and only if the ith element of the universe is in S. The vector can even represent a sequence of moves in a game or a path in a graph, where a_i contains the ith event in the sequence.

At each step in the backtracking algorithm, we try to extend a given partial solution $a = (a_1, a_2, ..., a_k)$ by adding another element at the end. After extending it, we must test whether what we now have is a solution: if so, we should print it or count it. If not, we must check whether the partial solution is still potentially extendible to some complete solution.

Backtracking constructs a tree of partial solutions, where each vertex represents a partial solution. There is an edge from x to y if node y was created by advancing from x. This tree of partial solutions provides an alternative way to think about backtracking, for the process of constructing the solutions corresponds exactly to doing a depth-first traversal of the backtrack tree. Viewing backtracking as a depth-first search on an implicit graph yields a natural recursive implementation of the basic algorithm.

```
Backtrack-DFS(A, k)
    if A = (a_1, a_2, ..., a_k) is a solution, report it.
    else
        k = k + 1
        compute S_k
        while S_k ≠ ∅ do
            a_k = an element in S_k
            S_k = S_k − a_k
            Backtrack-DFS(A, k)
```

Although a breadth-first search could also be used to enumerate solutions, a depth-first search is greatly preferred because it uses much less space. The current state of a search is completely represented by the path from the root to the current search depth-first node. This requires space proportional to the *height* of the tree. In breadth-first search, the queue stores all the nodes at the current level, which

is proportional to the *width* of the search tree. For most interesting problems, the width of the tree grows exponentially in its height.

Implementation

The honest working `backtrack` code is given below:

```
bool finished = FALSE;            /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
     int c[MAXCANDIDATES];     /* candidates for next position */
     int ncandidates;          /* next position candidate count */
     int i;                    /* counter */

     if (is_a_solution(a,k,input))
             process_solution(a,k,input);
     else {
             k = k+1;
             construct_candidates(a,k,input,c,&ncandidates);
             for (i=0; i<ncandidates; i++) {
                     a[k] = c[i];
                     make_move(a,k,input);
                     backtrack(a,k,input);
                     unmake_move(a,k,input);
                     if (finished) return;   /* terminate early */
             }
     }
}
```

Backtracking ensures correctness by enumerating all possibilities. It ensures efficiency by never visiting a state more than once.

Study how recursion yields an elegant and easy implementation of the backtracking algorithm. Because a new candidates array `c` is allocated with each recursive procedure call, the subsets of not-yet-considered extension candidates at each position will not interfere with each other.

The application-specific parts of this algorithm consists of five subroutines:

- `is_a_solution(a,k,input)` – This Boolean function tests whether the first k elements of vector a from a complete solution for the given problem. The last argument, `input`, allows us to pass general information into the routine. We can use it to specify n—the size of a target solution. This makes sense when constructing permutations or subsets of n elements, but other data may be relevant when constructing variable-sized objects such as sequences of moves in a game.

- `construct_candidates(a,k,input,c,ncandidates)` – This routine fills an array c with the complete set of possible candidates for the kth position of a, given the contents of the first $k-1$ positions. The number of candidates returned in this array is denoted by `ncandidates`. Again, `input` may be used to pass auxiliary information.

- `process_solution(a,k,input)` – This routine prints, counts, or however processes a complete solution once it is constructed.

- `make_move(a,k,input)` *and* `unmake_move(a,k,input)` – These routines enable us to modify a data structure in response to the latest move, as well as clean up this data structure if we decide to take back the move. Such a data structure could be rebuilt from scratch from the solution vector a as needed, but this is inefficient when each move involves incremental changes that can easily be undone.

  These calls function as null stubs in all of this section's examples, but will be employed in the Sudoku program of Section 7.3 (page 239).

We include a global `finished` flag to allow for premature termination, which could be set in any application-specific routine.

To really understand how backtracking works, you must see how such objects as permutations and subsets can be constructed by defining the right state spaces. Examples of several state spaces are described in subsections below.

7.1.1 Constructing All Subsets

A critical issue when designing state spaces to represent combinatorial objects is how many objects need representing. How many subsets are there of an n-element set, say the integers $\{1, \ldots, n\}$? There are exactly two subsets for $n = 1$, namely $\{\}$ and $\{1\}$. There are four subsets for $n = 2$, and eight subsets for $n = 3$. Each new element doubles the number of possibilities, so there are 2^n subsets of n elements.

Each subset is described by elements that are in it. To construct all 2^n subsets, we set up an array/vector of n cells, where the value of a_i (true or false) signifies whether the ith item is in the given subset. In the scheme of our general backtrack algorithm, $S_k = (true, false)$ and a is a solution whenever $k = n$. We can now construct all subsets with simple implementations of `is_a_solution()`, `construct_candidates()`, and `process_solution()`.

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);                /* is k == n? */
}
```

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
        c[0] = TRUE;
        c[1] = FALSE;
        *ncandidates = 2;
}


process_solution(int a[], int k)
{
        int i;                          /* counter */

        printf("{");
        for (i=1; i<=k; i++)
                if (a[i] == TRUE) printf(" %d",i);

        printf(" }\n");
}
```

Printing each out subset after constructing it proves to be the most complicated of the three routines!

Finally, we must instantiate the call to `backtrack` with the right arguments. Specifically, this means giving a pointer to the empty solution vector, setting $k = 0$ to denote that it is empty, and specifying the number of elements in the universal set:

```
generate_subsets(int n)
{
        int a[NMAX];                    /* solution vector */

        backtrack(a,0,n);
}
```

In what order will the subsets of $\{1, 2, 3\}$ be generated? It depends on the order of moves `construct_candidates`. Since *true* always appears before *false*, the subset of all trues is generated first, and the all-false empty set is generated last: {123}, {12}, {13}, {1}, {23}, {2}, {3}, {}

Trace through this example carefully to make sure you understand the backtracking procedure. The problem of generating subsets is more thoroughly discussed in Section 14.5 (page 452).

7.1.2 Constructing All Permutations

Counting permutations of $\{1, \ldots, n\}$ is a necessary prerequisite to generating them. There are n distinct choices for the value of the first element of a permutation. Once

we have fixed a_1, there are $n-1$ candidates remaining for the second position, since we can have any value except a_1 (repetitions are forbidden in permutation). Repeating this argument yields a total of $n! = \prod_{i=1}^{n} i$ distinct permutations.

This counting argument suggests a suitable representation. Set up an array/vector a of n cells. The set of candidates for the ith position will be the set of elements that have not appeared in the $(i-1)$ elements of the partial solution, corresponding to the first $i-1$ elements of the permutation.

In the scheme of the general backtrack algorithm, $S_k = \{1, \ldots, n\} - a$, and a is a solution whenever $k = n$:

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
        int i;                          /* counter */
        bool in_perm[NMAX];             /* who is in the permutation? */

        for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
        for (i=0; i<k; i++) in_perm[ a[i] ] = TRUE;

        *ncandidates = 0;
        for (i=1; i<=n; i++)
                if (in_perm[i] == FALSE) {
                        c[ *ncandidates] = i;
                        *ncandidates = *ncandidates + 1;
                }
}
```

Testing whether i is a candidate for the kth slot in the permutation can be done by iterating through all $k-1$ elements of a and verifying that none of them matched. However, we prefer to set up a bit-vector data structure (see Section 12.5 (page 385)) to maintain which elements are in the partial solution. This gives a constant-time legality check.

Completing the job requires specifying `process_solution` and `is_a_solution`, as well as setting the appropriate arguments to `backtrack`. All are essentially the same as for subsets:

```
process_solution(int a[], int k)
{
        int i;                          /* counter */

        for (i=1; i<=k; i++) printf(" %d",a[i]);

        printf("\n");
}
```

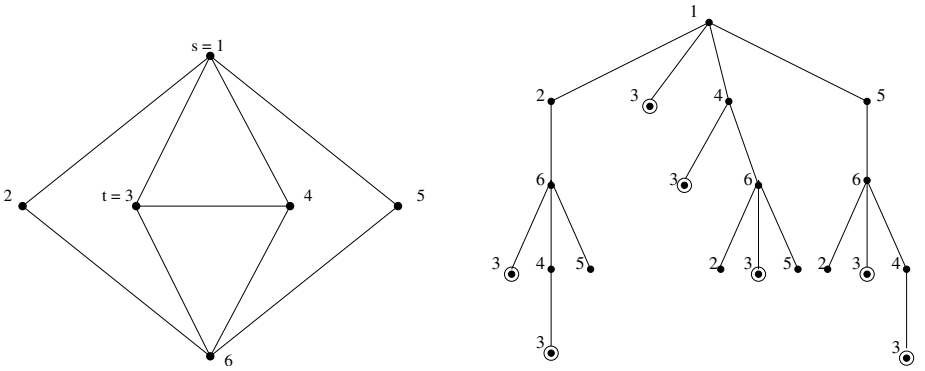

Figure 7.1: Search tree enumerating all simple s-t paths in the given graph (left).

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);
}


generate_permutations(int n)
{
        int a[NMAX];                    /* solution vector */

        backtrack(a,0,n);
}
```

As a consequence of the candidate order, these routines generate permutations in *lexicographic*, or sorted order—i.e., 123, 132, 213, 231, 312, and 321. The problem of generating permutations is more thoroughly discussed in Section 14.4 (page 448).

7.1.3 Constructing All Paths in a Graph

Enumerating all the simple s to t paths through a given graph is a more complicated problem than listing permutations or subsets. There is no explicit formula that counts the number of solutions as a function of the number of edges or vertices, because the number of paths depends upon the structure of the graph.

The starting point of any path from s to t is always s. Thus, s is the only candidate for the first position and $S_1 = \{s\}$. The possible candidates for the second position are the vertices v such that (s, v) is an edge of the graph, for the path wanders from vertex to vertex using edges to define the legal steps. In general,

S_{k+1} consists of the set of vertices adjacent to a_k that have not been used elsewhere in the partial solution A.

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
     int i;                    /* counters */
     bool in_sol[NMAX];        /* what's already in the solution? */
     edgenode *p;              /* temporary pointer */
     int last;                 /* last vertex on current path */

     for (i=1; i<NMAX; i++) in_sol[i] = FALSE;
     for (i=1; i<k; i++) in_sol[ a[i] ] = TRUE;

     if (k==1) {               /* always start from vertex 1 */
             c[0] = 1;
             *ncandidates = 1;
     }
     else    {
             *ncandidates = 0;
             last = a[k-1];
             p = g.edges[last];
             while (p != NULL) {
                     if (!in_sol[ p->y ]) {
                             c[*ncandidates] = p->y;
                             *ncandidates = *ncandidates + 1;
                     }
                     p = p->next;
             }
     }
}
```

We report a successful path whenever $a_k = t$.

```
is_a_solution(int a[], int k, int t)
{
        return (a[k] == t);
}

process_solution(int a[], int k)
{
        solution_count ++;        /* count all s to t paths */
}
```

The solution vector A must have room for all n vertices, although most paths are likely shorter than this. Figure 7.1 shows the search tree giving all paths from a particular vertex in an example graph.

7.2 Search Pruning

Backtracking ensures correctness by enumerating all possibilities. Enumerating all $n!$ permutations of n vertices of the graph and selecting the best one yields the correct algorithm to find the optimal traveling salesman tour. For each permutation, we could see whether all edges implied by the tour really exists in the graph G, and if so, add the weights of these edges together.

However, it would be wasteful to construct all the permutations first and then analyze them later. Suppose our search started from vertex v_1, and it happened that edge (v_1, v_2) was not in G. The next $(n-2)!$ permutations enumerated starting with (v_1, v_2) would be a complete waste of effort. Much better would be to prune the search after v_1, v_2 and continue next with v_1, v_3. By restricting the set of next elements to reflect only moves that are legal from the current partial configuration, we significantly reduce the search complexity.

Pruning is the technique of cutting off the search the instant we have established that a partial solution cannot be extended into a full solution. For traveling salesman, we seek the cheapest tour that visits all vertices. Suppose that in the course of our search we find a tour t whose cost is C_t. Later, we may have a partial solution a whose edge sum $C_A > C_t$. Is there any reason to continue exploring this node? No, because any tour with this prefix $a_1, \ldots, a_k$ will have cost greater than tour t, and hence is doomed to be nonoptimal. Cutting away such failed partial tours as soon as possible can have an enormous impact on running time.

Exploiting symmetry is another avenue for reducing combinatorial searches. Pruning away partial solutions identical to those previously considered requires recognizing underlying symmetries in the search space. For example, consider the state of our TSP search after we have tried all partial positions beginning with v_1. Does it pay to continue the search with partial solutions beginning with v_2? No. Any tour starting and ending at v_2 can be viewed as a rotation of one starting and ending at v_1, for these tours are cycles. There are thus only $(n-1)!$ distinct tours on n vertices, not $n!$. By restricting the first element of the tour to v_1, we save a factor of n in time without missing any interesting solutions. Detecting such symmetries can be subtle, but once identified they can usually be easily exploited.

Take-Home Lesson: Combinatorial searches, when augmented with tree pruning techniques, can be used to find the optimal solution of small optimization problems. How small depends upon the specific problem, but typical size limits are somewhere between $15 \leq n \leq 50$ items.

					1			2
			3		5			
		6					7	
7					3			
			4		8			
1								
			1	2				
		8					4	
		5				6		

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

Figure 7.2: Challenging Sudoku puzzle (l) with solution (r)

7.3 Sudoku

A Sudoku craze has swept the world. Many newspapers now publish daily Sudoku puzzles, and millions of books about Sudoku have been sold. British Airways sent a formal memo forbidding its cabin crews from doing Sudoku during takeoffs and landings. Indeed, I have noticed plenty of Sudoku going on in the back of my algorithms classes during lecture.

What is Sudoku? In its most common form, it consists of a 9×9 grid filled with blanks and the digits 1 to 9. The puzzle is completed when every row, column, and sector (3×3 subproblems corresponding to the nine sectors of a tic-tac-toe puzzle) contain the digits 1 through 9 with no deletions or repetition. Figure 7.2 presents a challenging Sudoku puzzle and its solution.

Backtracking lends itself nicely to the problem of solving Sudoku puzzles. We will use the puzzle here to better illustrate the algorithmic technique. Our state space will be the sequence of open squares, each of which must ultimately be filled in with a number. The candidates for open squares (i, j) are exactly the integers from 1 to 9 that have not yet appeared in row i, column j, or the 3×3 sector containing (i, j). We backtrack as soon as we are out of candidates for a square.

The solution vector `a` supported by `backtrack` only accepts a single integer per position. This is enough to store contents of a board square (1-9) but not the coordinates of the board square. Thus, we keep a separate array of `move` positions as part of our `board` data type provided below. The basic data structures we need to support our solution are:

```
#define DIMENSION 9                  /* 9*9 board */
#define NCELLS  DIMENSION*DIMENSION /* 81 cells in a 9*9 problem */

typedef struct {
    int x, y;                        /* x and y coordinates of point */
} point;
```

```
typedef struct {
    int m[DIMENSION+1][DIMENSION+1]; /* matrix of board contents */
    int freecount;                   /* how many open squares remain? */
    point move[NCELLS+1];            /* how did we fill the squares? */
} boardtype;
```

Constructing the candidates for the next solution position involves first picking the open square we want to fill next (`next_square`), and then identifying which numbers are candidates to fill that square (`possible_values`). These routines are basically bookkeeping, although the subtle details of how they work can have an enormous impact on performance.

```
construct_candidates(int a[], int k, boardtype *board, int c[],
                     int *ncandidates)
{
   int x,y;                     /* position of next move */
   int i;                       /* counter */
   bool possible[DIMENSION+1]; /* what is possible for the square */

   next_square(&x,&y,board);   /* which square should we fill next? */

   board->move[k].x = x;       /* store our choice of next position */
   board->move[k].y = y;

   *ncandidates = 0;

   if ((x<0) && (y<0)) return; /* error condition, no moves possible */

   possible_values(x,y,board,possible);
   for (i=0; i<=DIMENSION; i++)
           if (possible[i] == TRUE) {
                   c[*ncandidates] = i;
                   *ncandidates = *ncandidates + 1;
           }
}
```

We must update our `board` data structure to reflect the effect of filling a candidate value into a square, as well as remove these changes should we backtrack away from this position. These updates are handled by `make_move` and `unmake_move`, both of which are called directly from `backtrack`:

```
make_move(int a[], int k, boardtype *board)
{
        fill_square(board->move[k].x,board->move[k].y,a[k],board);
}


unmake_move(int a[], int k, boardtype *board)
{
        free_square(board->move[k].x,board->move[k].y,board);
}
```

One important job for these board update routines is maintaining how many free squares remain on the board. A solution is found when there are no more free squares remaining to be filled:

```
is_a_solution(int a[], int k, boardtype *board)
{
        if (board->freecount == 0)
                return (TRUE);
        else
                return(FALSE);
}
```

We print the configuration and turn off the backtrack search by setting off the global `finished` flag on finding a solution. This can be done without consequence because "official" Sudoku puzzles are only allowed to have one solution. There can be an enormous number of solutions to nonofficial Sudoku puzzles. Indeed, the empty puzzle (where no number is initially specified) can be filled in exactly 6,670,903,752,021,072,936,960 ways. We can ensure we don't see all of them by turning off the search:

```
process_solution(int a[], int k, boardtype *board)
{
        print_board(board);
        finished = TRUE;
}
```

This completes the program modulo details of identifying the next open square to fill (`next_square`) and identifying the candidates to fill that square (`possible_values`). Two reasonable ways to select the next square are:

- *Arbitrary Square Selection* – Pick the first open square we encounter, possibly picking the first, the last, or a random open square. All are equivalent in that there seems to be no reason to believe that one heuristic will perform any better than the other.

- *Most Constrained Square Selection* – Here, we check each of the open squares (i, j) to see how many number candidates remain for each—i.e. , have not already been used in either row i, column j, or the sector containing (i, j). We pick the square with the fewest number of candidates.

Although both possibilities work correctly, the second option is much, much better. Often there will be open squares with only one remaining candidate. For these, the choice is forced. We might as well make it now, especially since pinning this value down will help trim the possibilities for other open squares. Of course, we will spend more time selecting each candidate square, but if the puzzle is easy enough we may never have to backtrack at all.

If the most constrained square has two possibilities, we have a 1/2 probability of guessing right the first time, as opposed to a $(1/9)^{th}$ probability for a completely unconstrained square. Reducing our average number of choices from (say) 3 per square to 2 per square is an enormous win, since it multiplies for each position. If we have (say) 20 positions to fill, we must enumerate only $2^{20} = 1,048,576$ solutions. A branching factor of 3 at each of the 20 positions will result in over 3,000 times as much work!

Our final decision concerns the `possible_values` we allow for each square. We have two possibilities:

- *Local Count* – Our backtrack search works correctly if the routine generating candidates for board position (i, j) (`possible_values`) does the obvious thing and allows all numbers 1 to 9 that have not appeared in the given row, column, or sector.

- *Look ahead* – But what if our current partial solution has some *other* open square where there are no candidates remaining under the local count criteria? There is no possible way to complete this partial solution into a full Sudoku grid. Thus there *really* are zero possible moves to consider for (i, j) because of what is happening elsewhere on the board!

  We will discover this obstruction eventually, when we pick this square for expansion, discover it has no moves, and then have to backtrack. But why wait, since all our efforts until then will be wasted? We are *much* better off backtracking immediately and moving on.[1]

Successful pruning requires looking ahead to see when a solution is doomed to go nowhere, and backing off as soon as possible.

Table 7.1 presents the number of calls to `is_a_solution` for all four backtracking variants on three Sudoku instances of varying complexity:

[1]This look-ahead condition might have naturally followed from the most-constrained square selection, had it been permitted to select squares with no moves. However, my implementation credited squares that already contained numbers as having no moves, thus limiting the next square choices to squares with at least one move.

Pruning Condition		Puzzle Complexity		
`next_square`	`possible_values`	Easy	Medium	Hard
arbitrary	local count	1,904,832	863,305	never finished
arbitrary	look ahead	127	142	12,507,212
most constrained	local count	48	84	1,243,838
most constrained	look ahead	48	65	10,374

Table 7.1: Sudoku run times (in number of steps) under different pruning strategies

- The *Easy* board was intended to be easy for a human player. Indeed, my program solved it without any backtracking steps when the most constrained square was selected as the next position.
- The *Medium* board stumped all the contestants at the finals of the World Sudoku Championship in March 2006. The decent search variants still required only a few backtrack steps to dispatch this problem.
- The *Hard* problem is the board displayed in Figure 7.2, which contains only 17 fixed numbers. This is the fewest specified known number of positions in any problem instance that has only one complete solution.

What is considered to be a "hard" problem instance depends upon the given heuristic. Certainly you know people who find math/theory harder than programming and others who think differently. Heuristic A may well think instance I_1 is easier than I_2, while heuristic B ranks them in the other order.

What can we learn from these experiments? Looking ahead to eliminate dead positions as soon as possible is the best way to prune a search. Without this operation, we never finished the hardest puzzle and took thousands of times longer than we should have on the easier ones.

Smart square selection had a similar impact, even though it nominally just rearranges the order in which we do the work. However, doing more constrained positions first is tantamount to reducing the outdegree of each node in the tree, and each additional position we fix adds constraints that help lower the degree for future selections.

It took the better part of an hour (48:44) to solve the puzzle in Figure 7.2 when I selected an arbitrary square for my next move. Sure, my program was faster in most instances, but Sudoku puzzles are designed to be solved by people using pencils in much less time than this. Making the next move in the most constricted square reduced search time by a factor of over 1,200. Each puzzle we tried can now be solved in seconds—the time it takes to reach for the pencil if you want to do it by hand.

This is the power of a pruning search. Even simple pruning strategies can suffice to reduce running time from impossible to instantaneous.

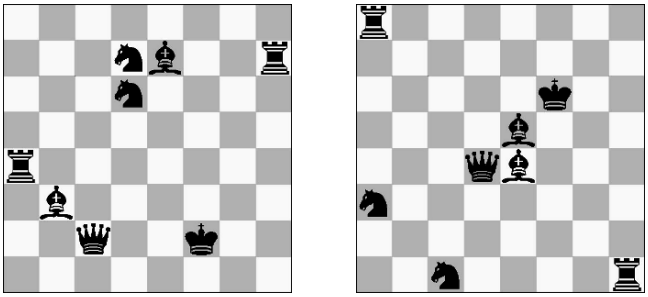

Figure 7.3: Configurations covering 63 but not 64 squares

7.4 War Story: Covering Chessboards

Every researcher dreams of solving a classical problem—one that has remained open and unsolved for over a century. There is something romantic about communicating across the generations, being part of the evolution of science, and helping to climb another rung up the ladder of human progress. There is also a pleasant sense of smugness that comes from figuring out how to do something that nobody could do before you.

There are several possible reasons why a problem might stay open for such a long period of time. Perhaps it is so difficult and profound that it requires a uniquely powerful intellect to solve. A second reason is technological—the ideas or techniques required to solve the problem may not have existed when it was first posed. A final possibility is that no one may have cared enough about the problem in the interim to seriously bother with it. Once, I helped solve a problem that had been open for over a hundred years. Decide for yourself which reason best explains why.

Chess is a game that has fascinated mankind for thousands of years. In addition, it has inspired many combinatorial problems of independent interest. The combinatorial explosion was first recognized with the legend that the inventor of chess demanded as payment one grain of rice for the first square of the board, and twice as much for the $(i+1)$st square than the ith square. The king was astonished to learn he had to cough up $\sum_{i=1}^{64} 2^i = 2^{65} - 1 = 36{,}893{,}488{,}147{,}419{,}103{,}231$ grains of rice. In beheading the inventor, the wise king first established pruning as a technique for dealing with the combinatorial explosion.

In 1849, Kling posed the question of whether all 64 squares on the board can be simultaneously threatened by an arrangement of the eight main pieces on the chess board—the king, queen, two knights, two rooks, and two oppositely colored bishops. Pieces do not threaten the square they sit on. Configurations that simultaneously threaten 63 squares, such as those in Figure 7.3, have been known for a long time, but whether this was the best possible remained an open problem. This problem

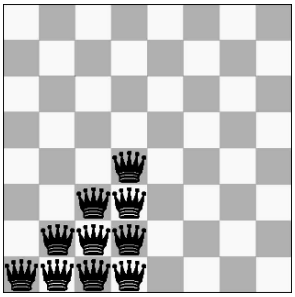

Figure 7.4: The ten unique positions for the queen, with respect to symmetry

seemed ripe for solution by exhaustive combinatorial searching, although whether it was solvable depended upon the size of the search space.

Consider the eight main pieces in chess (king, queen, two rooks, two bishops, and two knights). How many ways can they be positioned on a chessboard? The trivial bound is $64!/(64-8)! = 178{,}462{,}987{,}637{,}760 \approx 10^{15}$ positions. Anything much larger than about 10^9 positions would be unreasonable to search on a modest computer in a modest amount of time.

Getting the job done would require significant pruning. Our first idea was to remove symmetries. Accounting for orthogonal and diagonal symmetries left only ten distinct positions for the queen, shown in Figure 7.4.

Once the queen is placed, there are $64 \cdot 63/2 = 2{,}016$ distinct ways to position a pair of rooks or knights, 64 places to locate the king, and 32 spots for each of the white and black bishops. Such an exhaustive search would test $2{,}663{,}550{,}812{,}160 \approx 10^{13}$ distinct positions—still much too large to try.

We could use backtracking to construct all of the positions, but we had to find a way to prune the search space significantly. Pruning the search meant that we needed a quick way to prove that there was no way to complete a partially filled-in position to cover all 64 squares. Suppose we had already placed seven pieces on the board, and together they covered all but 10 squares of the board. Say the remaining piece was the king. Can there be a position to place the king so that all squares are threatened? The answer must be no, because the king can threaten at most eight squares according to the rules of chess. There can be no reason to test any king position. We might win big pruning such configurations.

This pruning strategy required carefully ordering the evaluation of the pieces. Each piece can threaten a certain maximum number of squares: the queen 27, the king/knight 8, the rook 14, and the bishop 13. We would want to insert the pieces in decreasing order of mobility. We can prune when the number of unthreatened squares exceeds the sum of the maximum coverage of the unplaced pieces. This sum is minimized by using the decreasing order of mobility.

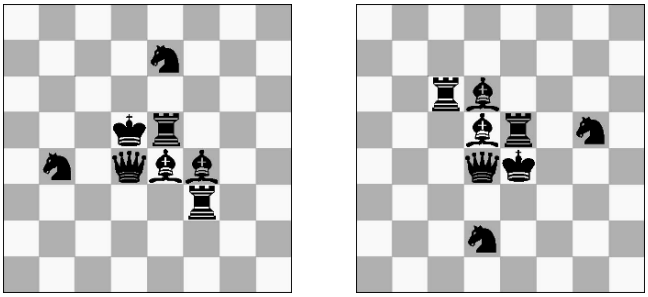

Figure 7.5: Weakly covering 64 squares

When we implemented a backtrack search using this pruning strategy, we found that it eliminated over 95% of the search space. After optimizing our move generation, our program could search over 1,000 positions per second. But this was still too slow, for $10^{12}/10^3 = 10^9$ seconds meant 1,000 days! Although we might further tweak the program to speed it up by an order of magnitude or so, what we really needed was to find a way to prune more nodes.

Effective pruning meant eliminating large numbers of positions at a single stroke. Our previous attempt was too weak. What if instead of placing up to eight pieces on the board simultaneously, we placed *more* than eight pieces. Obviously, the more pieces we placed simultaneously, the more likely they would threaten all 64 squares. But *if* they didn't cover, all subsets of eight distinct pieces from the set couldn't possibly threaten all squares. The potential existed to eliminate a vast number of positions by pruning a single node.

So in our final version, the nodes of our search tree corresponded to chessboards that could have any number of pieces, and more than one piece on a square. For a given board, we distinguished *strong* and *weak* attacks on a square. A strong attack corresponds to the usual notion of a threat in chess. A square is *weakly attacked* if the square is strongly attacked by some subset of the board—that is, a weak attack ignores any possible blocking effects of intervening pieces. All 64 squares can be weakly attacked with eight pieces, as shown in Figure 7.5.

Our algorithm consisted of two passes. The first pass listed boards where every square was weakly attacked. The second pass filtered the list by considering blocking pieces. A weak attack is much faster to compute (no blocking to worry about), and any strong attack set is always a subset of a weak attack set. The position could be pruned whenever there was a non-weakly threatened square.

This program was efficient enough to complete the search on a slow 1988-era IBM PC-RT in under one day. It did not find a single position covering all 64 squares with the bishops on opposite colored squares. However, our program showed that it is possible to cover the board with *seven* pieces provided a queen and a knight can occupy the same square, as shown in Figure 7.6.

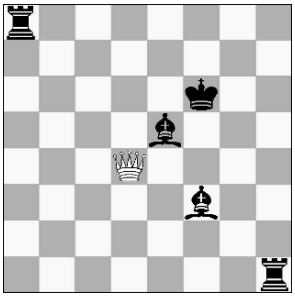

Figure 7.6: Seven pieces suffice when superimposing queen and knight (shown as a white queen)

Take-Home Lesson: Clever pruning can make short work of surprisingly hard combinatorial search problems. Proper pruning will have a greater impact on search time than any other factor.

7.5 Heuristic Search Methods

Heuristic methods provide an alternate way to approach difficult combinatorial optimization problems. Backtracking gave us a method to find the best of all possible solutions, as scored by a given objective function. However, any algorithm searching all configurations is doomed to be impossible on large instances.

In this section, we discuss such heuristic search methods. We devote the bulk of our attention to simulated annealing, which I find to be the most reliable method to apply in practice. Heuristic search algorithms have an air of voodoo about them, but how they work and why one method might work better than another follows logically enough if you think them through.

In particular, we will look at three different heuristic search methods: random sampling, gradient-descent search, and simulated annealing. The traveling salesman problem will be our ongoing example for comparing heuristics. All three methods have two common components:

- *Solution space representation* – This is a complete yet concise description of the set of possible solutions for the problem. For traveling salesman, the solution space consists of $(n-1)!$ elements—namely all possible circular permutations of the vertices. We need a data structure to represent each element of the solution space. For TSP, the candidate solutions can naturally be represented using an array S of $n-1$ vertices, where S_i defines the $(i+1)$st vertex on the tour starting from v_1.
- *Cost function* – Search methods need a *cost* or *evaluation* function to access the quality of each element of the solution space. Our search heuristic

identifies the element with the best possible score—either highest or lowest depending upon the nature of the problem. For TSP, the cost function for evaluating a given candidate solution S should just sum up the costs involved, namely the weight of all edges (S_i, S_{i+1}), where S_{n+1} denotes v_1.

7.5.1 Random Sampling

The simplest method to search in a solution space uses random sampling. It is also called the *Monte Carlo method*. We repeatedly construct random solutions and evaluate them, stopping as soon as we get a good enough solution, or (more likely) when we are tired of waiting. We report the best solution found over the course of our sampling.

True random sampling requires that we are able to select elements from the solution space *uniformly at random*. This means that each of the elements of the solution space must have an equal probability of being the next candidate selected. Such sampling can be a subtle problem. Algorithms for generating random permutations, subsets, partitions, and graphs are discussed in Sections 14.4-14.7.

```
random_sampling(tsp_instance *t, int nsamples, tsp_solution *bestsol)
{
        tsp_solution s;                 /* current tsp solution */
        double best_cost;               /* best cost so far */
        double cost_now;                /* current cost */
        int i;                          /* counter */

        initialize_solution(t->n,&s);
        best_cost = solution_cost(&s,t);
        copy_solution(&s,bestsol);

        for (i=1; i<=nsamples; i++) {
                random_solution(&s);
                cost_now = solution_cost(&s,t);
                if (cost_now < best_cost) {
                        best_cost = cost_now;
                        copy_solution(&s,bestsol);
                }
        }
}
```

When might random sampling do well?

- *When there are a high proportion of acceptable solutions* – Finding a piece of hay in a haystack is easy, since almost anything you grab is a straw. When solutions are plentiful, a random search should find one quickly.

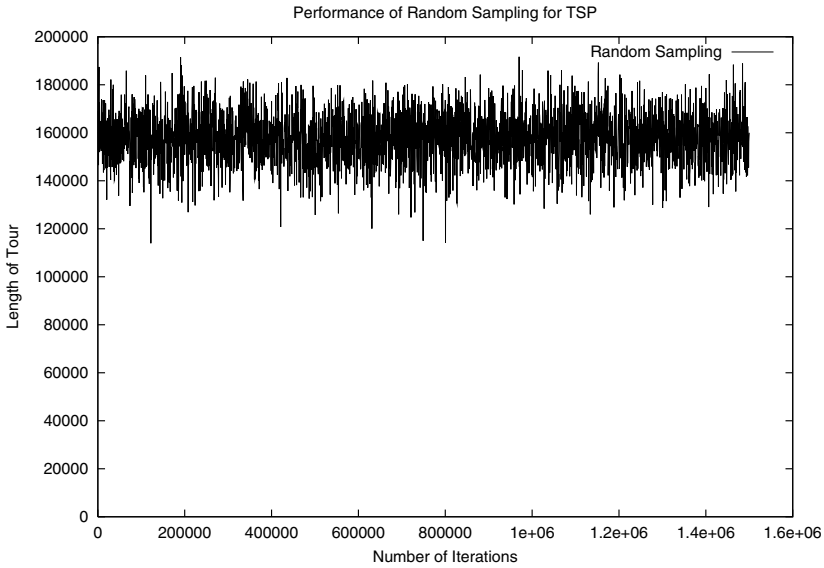

Figure 7.7: Search time/quality tradeoffs for TSP using random sampling.

Finding prime numbers is domain where a random search proves successful. Generating large random prime numbers for keys is an important aspect of cryptographic systems such as RSA. Roughly one out of every $\ln n$ integers are prime, so only a modest number of samples need to be taken to discover primes that are several hundred digits long.

- *When there is no coherence in the solution space* – Random sampling is the right thing to do when there is no sense of when we are getting *closer* to a solution. Suppose you wanted to find one of your friends who has a social security number that ends in 00. There is not much you can hope to do but tap an arbitrary fellow on their shoulder and ask. No cleverer method will be better than random sampling.

  Consider again the problem of hunting for a large prime number. Primes are scattered quite arbitrarily among the integers. Random sampling is as good as anything else.

How does random sampling do on TSP? Pretty lousy. The best solution I found after testing 1.5 million random permutations of a classic TSP instance (the capital cities of the 48 continental United States) was 101,712.8. This is more than three times the cost of the optimal tour! The solution space consists almost entirely

of mediocre to bad solutions, so quality grows very slowly with the amount of sampling / running time we invest. Figure 7.7 presents the arbitrary up-and-down movements of random sampling and generally poor quality solutions encountered on the journey, so you can get a sense of how the score varied over each iteration.

Most problems we encounter, like TSP, have relatively few good solutions but a highly coherent solution space. More powerful heuristic search algorithms are required to deal effectively with such problems.

Stop and Think: Picking the Pair

Problem: We need an efficient and unbiased way to generate random pairs of vertices to perform random vertex swaps. Propose an efficient algorithm to generate elements from the $\binom{n}{2}$ *unordered* pairs on $\{1, \ldots, n\}$ uniformly at random.

Solution: Uniformly generating random structures is a surprisingly subtle problem. Consider the following procedure to generate random unordered pairs:

```
i = random_int(1,n-1);
j = random_int(i+1,n);
```

It is clear that this indeed generates unordered pairs, since $i < j$. Further, it is clear that all $\binom{n}{2}$ unordered pairs can indeed be generated, assuming that `random_int` generates integers uniformly between its two arguments.

But are they uniform? The answer is no. What is the probability that pair $(1, 2)$ is generated? There is a $1/(n-1)$ chance of getting the 1, and then a $1/(n-1)$ chance of getting the 2, which yields $p(1, 2) = 1/(n-1)^2$. But what is the probability of getting $(n-1, n)$? Again, there is a $1/n$ chance of getting the first number, but now there is only one possible choice for the second candidate! This pair will occur n times more often than the first!

The problem is that fewer pairs start with big numbers than little numbers. We could solve this problem by calculating exactly how unordered pairs start with i (exactly $(n-i)$) and appropriately bias the probability. The second value could then be selected uniformly at random from $i+1$ to n.

But instead of working through the math, let's exploit the fact that randomly generating the n^2 *ordered* pairs uniformly is easy. Just pick two integers independently of each other. Ignoring the ordering (i.e. , permuting the ordered pair to unordered pair (x, y) so that $x < y$) gives us a $2/n^2$ probability of generating each unordered pair of distinct elements. If we happen to generate a pair (x, x), we discard it and try again. We will get unordered pairs uniformly at random in constant expected time using the following algorithm:

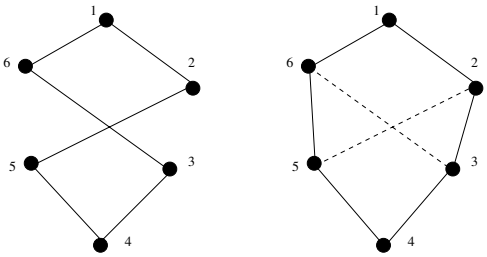

Figure 7.8: Improving a TSP tour by swapping vertices 2 and 6

```
do {
     i = random_int(1,n);
     j = random_int(1,n);
     if (i > j) swap(&i,&j);
} while (i==j);
```

■

7.5.2 Local Search

Now suppose you want to hire an algorithms expert as a consultant to solve your problem. You *could* dial a phone number at random, ask if they are an algorithms expert, and hang up the phone if they say no. After many repetitions you will probably find one, but it would probably be more efficient to ask the fellow on the phone for someone more likely to know an algorithms expert, and call *them* up instead.

A local search employs *local neighborhood* around every element in the solution space. Think of each element x in the solution space as a vertex, with a directed edge (x, y) to every candidate solution y that is a neighbor of x. Our search proceeds from x to the most promising candidate in x's neighborhood.

We certainly do *not* want to explicitly construct this neighborhood graph for any sizable solution space. Think about TSP, which will have $(n-1)!$ vertices in this graph. We are conducting a heuristic search precisely because we cannot hope to do this many operations in a reasonable amount of time.

Instead, we want a general transition mechanism that takes us to the next solution by slightly modifying the current one. Typical transition mechanisms include swapping a random pair of items or changing (inserting or deleting) a single item in the solution.

The most obvious transition mechanism for TSP would be to swap the current tour positions of a random pair of vertices S_i and S_j, as shown in Figure 7.8.

This changes up to eight edges on the tour, deleting the edges currently adjacent to both S_i and S_j, and adding their replacements. Ideally, the effect that these incremental changes have on measuring the quality of the solution can be computed incrementally, so cost function evaluation takes time proportional to the size of the change (typically constant) instead of linear to the size of the solution.

A local search heuristic starts from an arbitrary element of the solution space, and then scans the neighborhood looking for a favorable transition to take. For TSP, this would be *transition*, which lowers the cost of the tour. In a *hill-climbing* procedure, we try to find the top of a mountain (or alternately, the lowest point in a ditch) by starting at some arbitrary point and taking any step that leads in the direction we want to travel. We repeat until we have reached a point where all our neighbors lead us in the wrong direction. We are now *King of the Hill* (or *Dean of the Ditch*).

We are probably not *King of the Mountain*, however. Suppose you wake up in a ski lodge, eager to reach the top of the neighboring peak. Your first transition to gain altitude might be to go upstairs to the top of the building. And then you are trapped. To reach the top of the mountain, you must go downstairs and walk outside, but this violates the requirement that each step has to increase your score. Hill-climbing and closely related heuristics such as greedy search or gradient descent search are great at finding local optima quickly, but often fail to find the globally best solution.

```
hill_climbing(tsp_instance *t, tsp_solution *s)
{
     double cost;                /* best cost so far */
     double delta;               /* swap cost */
     int i,j;                    /* counters */
     bool stuck;                 /* did I get a better solution? */
     double transition();

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

     do {
             stuck = TRUE;
             for (i=1; i<t->n; i++)
                     for (j=i+1; j<=t->n; j++) {
                             delta = transition(s,t,i,j);
                             if (delta < 0) {
                                     stuck = FALSE;
                                     cost = cost + delta;
                             }
```