Combinatorial Search

June 18, 2020

1 Backtracking

Backtracking is a technique for listing all possible solutions for a combinatorial algorithm problem.

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*)

What these problems have in common is that we must generate each one possible configuration exactly once. Avoiding both repetition 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 sequwnce 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 potentiall extendable to some complete solution

The vector can even represent a sequwnce of moves in a game or a path in a graph, where $ a_i $ contains the $ ith $ event in the sequence

Backtracking constructs a tree of partial solutions, where each vertex represents a partial solution. There is an edfe 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 algortihm

```
Backtrack-DFS(A,k)
if A = (a1,a2,...,ak) is a solution, report it.
else
    k = k + 1
    compute Sk
    while Sk !=  do
```

```
        ak = an element in Sk
        Sk = Sk - ak
        Backtrack-DFS(A,k)
```

DFS is perfered over BFS because of the space saved. The curent 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 BFS, the queue stores all the nodes at the current height, which is proportional to the width of the search tree. The tree grows exponentially in its widith compared to its height

Backtracking ensures correctness by enumerating all possibilites. It ensures effciency by never visiting a state more than once

1.1 Implementation

```
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 */
        }
    }
}
```

- `is_a_solution(a, k, input)`: the Boolean function tests whether the first $ k $ element is a vector $ a $ from a complete solution for the given problem. **input**, allows us to pass general information into the routine. We can use it to specifiy $ n $-the size of a target solution. This makes sense when consturcting permutations or subsets of $ n $ elements, but other data may be relevent when constructing variable-sized objects such as sequences of moves in a game
- `construct_candidates(a, k, input, c, ncandidates)`: this routine or policy fills an array $ c $ with the complete set of possible candiates for the $ kth $ position of $ a $, given the contents of the first $ k - 1 $ positions. The number of candidates returns in the array is denoted by `ncandidates`. Again **input** maybe 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 lastest move, as well as clean up this data structre if we decide to take back the move. Such a data structure could be rebuilt from scratc from the solution vector $ a $ as needed, but this is inefficient when each move involved incremental changes that can easily be undone

1.2 Constructing All Subsets

Problem with combinatorial objects is how many objects need representing. Each new element doubles the number of possibilities, so there are $ 2^n $ subsets of $ n $ elements

To consturct all $ 2^n $ subsets, we set up an array 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 solutuon whenever $ k = n $

```
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");
}
```

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

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

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

What order will subset of \$ {1, 2, 3 } \$ be generated?

- \$ (123), (12), (13), (1), (23), (2), (3), () \$

1.3 Constructing All Permutations

Counting permutations of \$ 1,..., n \$ is a necessary prerequsite to generating them. There are \$ n \$ distincy choices for the value of the first element of a permutation.

Once we have a fixed \$ a_1 \$, there are \$ n - 1 \$ candiates 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! \$ distinct permutations

Set up an array \$ a \$ of \$ n \$ cells. The set of candiates 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 genral backtracking algortihms, \$ S_k = (1, ..., 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. Howerver, we perfer to set up a bit-vector data structure to maintain which elements are in the partial solution. This gives a constant-time legality check

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

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

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

    printf("\n");
}

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);
}
```

Solution to $ 123$ is - $ 123, 132, 213, 312, 321 $

1.4 Constructing All Paths in a Graph

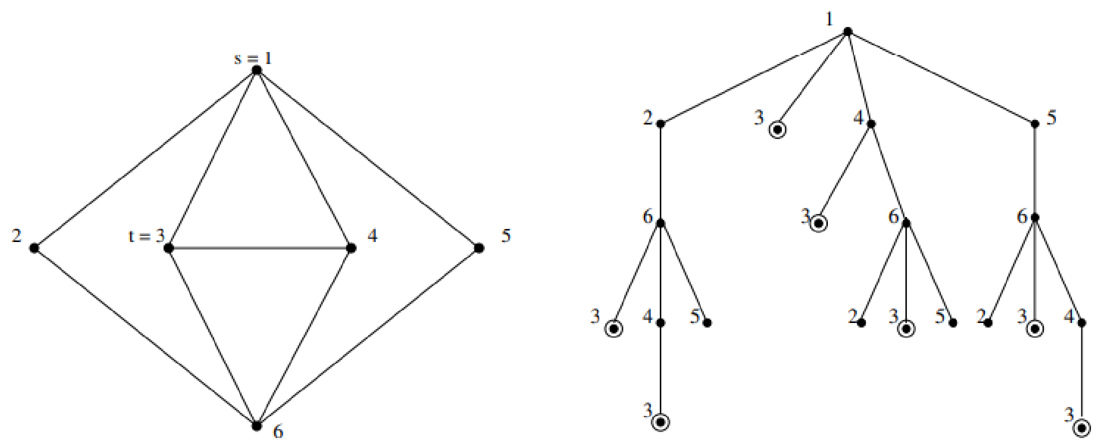


Figure 1: Search tree enumerating all simple $ s- t $ paths in the given graph (left).

Enumerating all the simple $ s $ to $ t $ paths through a given graph is more complicated problem than listing permutations or subsets.

There is no exlicit formula that counts the number of solutions because the number of solutions depends upon the structure of the graph

The starting point of ant path from $ s $ to $ t $ is always $ s $. Thus $ S_1 = (s) $. The possible candiates 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} $ conssits 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, althoug most paths are likely shorter than this. Figure 1 shows the search tree giving all paths form a particular vertex in an example graph

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 yeilds the correct algorithm to find the optimal traveling salesman tour.

For each permutation, we could see wether all edges implied by the tour really exists in the graph $ G $, and if so, add the weights of these edges together

It would be a 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 we would be to prune the search after $ v_1, v_2 $ and continue with $ v_1, v_3 $. By restricting th 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 curring off the search the instant we have established that a partial solution cannot be extended into full solution

For the traveling salesman problem we seek the cheapest tour that visits all vertices. Suppose that in the course of our search we find a tour $ t $ whos cost is $ C_t $. Later, we may have a partial solution $ a $ whos edge sum $ C_A > C_t.

Is there any reason to continue exploring this node? No, because any tour with the prefix $ a_1, ..., a_k $ will have cost greater than tour $ t $, and hence is doomed to be non-optimal

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.

Consider the state our Traveling Salesmen Problem stearch after we have tried all partial positions beginning with $ v_1 $. Does it pay to continue the search with partial solution 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 intersting solutions

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

3 Sudoku

							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 2: Challenging Sudoku puzzle $ (1) $ with solution $ (r) $

We will use the puzzle here to illustrate an algorithmic technique of backtracking

Our state space will be the sequence of open squares, each of which must ultimately be filled in with a number. The candiates for open squares $ (i, j) $ are exactly the integers from $ 1 $ to $ 9 $ that have not yey appeared in row $ i $, column $ j $, or the $ 3 * 3 $ sector containing $ (i, j) $.

We backtrack as soon as we are out o candiates 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 seprate array of **move** positions as part of our **board** data type provided below

The data structure needed is below

```
#define DIMENSION 9 /* 9*9 board */
#define NCELLS DIMENSION*DIMENSION /* 81 cells in a 9*9 problem */
typedef struct {
    int x, y;
} 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 candiates for the next solution prosition involves first picking the open square we want to fill next (`next_square`), and then identifying which numbers are candiates to fill that square (`possible_value`). These routines are basically bookkeeping, although the subtle details of how they work can have an enormous impact on performace

```
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 candiate 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 wich 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 maintained 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);
}
```

Offical Sudoku puzzles are only allowed to have one solution. There can be enormous number of solutions to nonoffical Sudoku puzzles. Indeed, the empty puzzle (where no number is initiall specified) can be filled a god teir number of ways. We can ensure we dont see them by turning off search once we solved the board

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

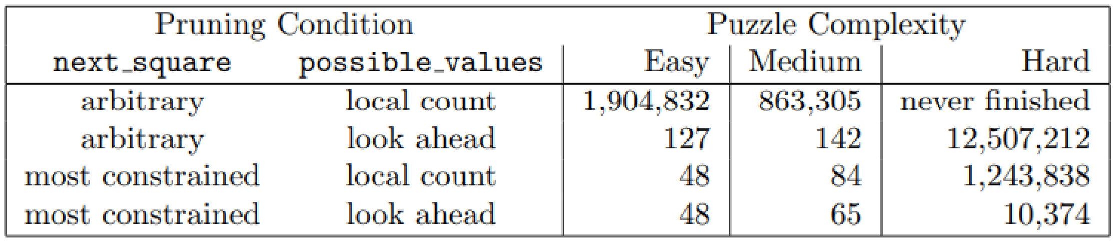
Square Selection Method - Arbitrary Square Selection: Pick the first open square we encounter, possibly picking the first, the last or a random open square. ALl are equivalent in that there seems to be no reason to belive that one heuristic will perform any better than the other - Most Constrained Square Selection: Here, we check each of the open squares \$ (i, j)\$ to see how many number of 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 candiates

Second Option is better, because it reduces possibilites

If the most constrained square has two possibilites, we have at \$ 1/2 \$ probability of guessing right the first time, as opposed to a \$ (1/9)^th \$ probability for a completely unconstrained square.

Value Selection Method - Local Count: Our backtrack search works correctly if the routine generating candidated for board position \$ (i, j) \$ (`possible_values`) does the obvious thing and allows all numbers from \$ 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 othe ropen square where there are no candidates remaining under the local count criteria? There is no possible way to complete this parital solution in a full Sudoku gird. Thus there really are zero possible moves to consider for $ (i,j) $ becuase of what is happening elsewhere on the board! We are better off backtracking immediately and moving head

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



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

Figure 3: Sudoku run times (in number of steps) under different pruning stragtegies

Pruning Has Magic Powers: If we select the most constrained square with look ahead, we can finish Hard puzzles in a fraction of the time without these prunings

4 Heuristic Search Methods

Heuristics provide an alternative way to approach difficult combinatorial optimiaztion problems

Backtracking gives us a method to find the best of all possible solutions, as scored by a given objective function. Howerver, any algorithm searching all configurations is doomed to be impossible on large instances

The Three different heurisitic search methods are: **random sampling**, **gradient-descent search** and **simulated annealing**

4.0.1 Properties of Heuristics

- Solution space representation
    - This is a complete yet concise description of the set of possible solutions for the problem. For the traveling salesman, the solution space consists of $ (n- 1)! $ elements- namely all possible circular permutations of the vertices. We usually need a data structure to represent each element of the solution space. For the traveling space salesman, the candidate solutions can be naturally be represented using an array $ S $ of $ n - 2 $ vertices, where $ S_i $ define 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 heuristic identifies the element with the best possible socre- either highest or lowest depending upon the nature of the problem. For the traveling salesman problem, the cost function for evaluating a given candidate solution $ S $ should just sum the costs involved, namely the weight of all edges $ (S_i, S_{i+1})$, where $ S_{n+1}$ denotes $ v_1 $

4.1 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 over the course of our sampling

True random sampling requres 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

```
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);
        }
    }
}
```

4.1.1 What might random sampling do well?

- When there are a high proportion of acceptable solutions
  - Finding s 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. Finding prime numbers is domain whenre a random search proves successufl. Generating large random prime numbers for key is important in cryptographic systems.
- 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 SSN that ends in 00. There is not much you can hope to do buy tap an arbitrary fellow on their shoulders and ask.

4.2 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 $ n/2 $

unordered pairs on {1,...,n} uniformly at random

Solution: Uniformly generating random structures is a surprisingly subtle problem.

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 a $ 2/n^2 $ propability of generating each unordered pair of distinct elements

If we denerate $ (x, x) $ we discard it and try again. We will get unordered pairs uniformly at random in constant expected time using the following algortihm

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

4.3 Local Search

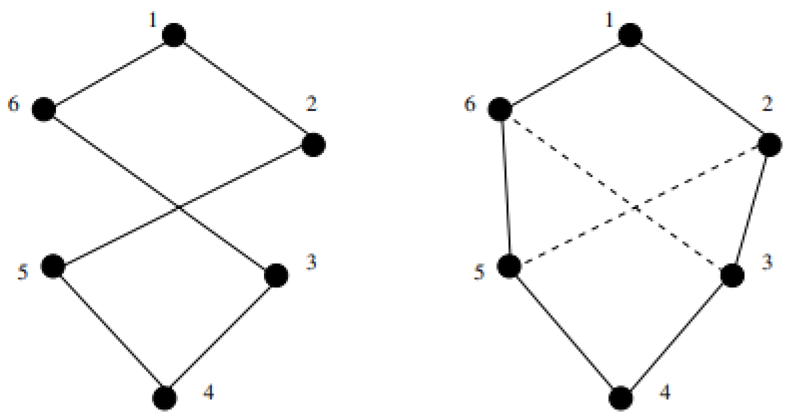


Figure 4 Improving a Traveling Salesman Problem tour by swapping vertices $ 2 $ and $ 6 $

Suppose you wanted to call a plummer. You could call random numbers, ask if they are plummers untill you find a plummer.

But if what if instead you call someone, ask if they know a plummer and then call that plumemr

A local search employes *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 want a general transition mechanism that takes us to the next solution by slightly modifying the current one. Typical transition mechanism include swapping a random pair of items or changing (insert/delete) a single item in the solution

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

For Traveling Salesman, our transition in the neightborhood, would be to find the edge that lowers the cost of the tour

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

4.3.1 What do local search do well?

- When there is great coherence in the solution space
  - Hill climbing is at its best when solution space is convex, meaning it consists of exactly one hill, no matter where you start on the hill, there is alway one direction you can go untill you are at the global maximum. Think binary search or think Maximum Stock Sell
- Whenever the cost of incremental evaluation is much cheaper than global evaluation
  - It costs \$ Θ(n) \$ to evaluate the cost of an arbitrary \$ n-vertex \$ candidate traveling sales man problem solution, because we must total the cost of each edge in the circular permutation describing the tour. Once that is found, howerver, the cost of the tour after swapping a given pair of vertices can be determined in constant time

Drawback of local search is that soon there isn't anything left for us to do as we find the local optimum

4.4 Simulated Annealing

Simulated annealing is a heuristic search procedure that allows occasional transitions leading to more expensive (and hence inferior) solutions

This keeps our search from getting stuck in a local optima.

In thermodynamics theory, the energy of a system is described by the energy state of each particle constituting it. A particles eanergy state jumps about randomply with such transitions govered by the tempature of the system

The probability of moving from a high-energy state to a lower-energy state is very high, but its still a non-zero 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 cocur

```
Simulated-Annealing()
    Create initial solution S
    Initialize temperature t
    repeat
        for i = 1 to iteration-length do
        Generate a random transition from S to Si
        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 = Si
    Reduce temperature t
    until (no change in C(S))
    Return S
```

Through random transitions generated according to the given probability distribution, we can mimic the physics to solve arbitrary combinatorial optimization problems

- **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.**

Initally we are eager to use randomness to explore the search space widely, so the probability of accepting a negative transition should be high, But as the search progresses, we seek to limit transitions to local improvements and optimizations.

Cooling Scheduling Parameters: - Inital state tempature is intially $ t__1 = 1 $ - Tempature decrement function: Typically $ t__k = a*t__{k-1}$. This implies an expodential decay in tempature, as oposed to a linear cecay - Number of iterations between tempature change- typically 100 to 1,000 iterations might be permitted before lowering the tempature - 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 − (C(s__i)−C(s__{i+1}))/ k*t__i >= r$, where r is a random number $ 0 <= 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, search is terminated and the current solution is reported

Anneling Solutions work very well it should be a method of choice

4.4.1 Implementation

```
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;
    }
}
```

4.4.2 Applications of Simulated Annealing

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 specifiy an electronic circuit, the maximum cut in the graph defines the largest amount of data communication that can take plave in the circuit simultaneously

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 neigthbors minus the wieght of the new neightbors. This can be computed in time proportional to the degree of the vertex

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

We seek a placement of rectangles that minimize area and wire lenght subject to the constraints that no two rectangles ovelap each other

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

5 Other Heuristic Search Methods

5.1 Genetic Algortims

Genetic algorithm report to work because they try to encapsulate randomness of evolution and biology but do not work

5.2 Parallel Algorithms

If you had more power, you could potientally just let it run

Problems: - There is often a small upper bound on the potential win. Sometimes its better to tweak the algortihm and gain massive improvements - Speedups mean nothing. Even if you have massive horse power, does not matter because again, good algortihms can still be faster - Parallel algorithms are tough to debug. Its not linear, how do you follow the algorithm and where its going