Simulated Annealing: A Heuristic Optimization Algorithm

Girish Keshav Palshikar

Abstract: A variety of practical problems search a finite (but very large) set of possible states to identify a state that has the lowest cost or an acceptably low cost. A systematic search to locate such a global minimum state is often intractable. Several heuristic search techniques have been designed for such problems. We examine a well-known probabilistic local search technique called simulated annealing (SA) that mimics the natural process of slow cooling of liquids that leads to a solid form that has the lowest energy. We provide a general C implementation of the SA algorithm and illustrate it by applying it to the problem of producing an aesthetic drawing of a given graph.

Introduction

There are many practical problems that need to find a “lowest-cost” solution that minimizes some given cost function. For example, in the famous traveling salesman problem (TSP), given a map connecting N cities through a road network, the task is to find a shortest length tour starting and ending at a given city and where the salesman visits every other city exactly once. The obvious cost here is the total distance traveled in a tour; there may be additional factors that contribute to evaluating the desirability of a proposed tour – e.g., shortest time to be spent in cold regions. As another example, the placement of components on a circuit board is expected to minimize the total length of the wire used and also the number of crossings.

These so-called combinatorial optimization problems have some common characteristics. There is a state-space in which the search for an optimal solution needs to be conducted. Each state (or configuration) in such a state-space is a candidate solution. For example, the state space for TSP consists of all possible tours in the given graph and the state-space for the circuit placement problem consists of all possible placements of given components in a given m × n grid, subject to connectivity in the circuit diagram. The state space is often finite, though very large and is characterized in terms of a small number of parameters where each parameter has a finite domain of possible values. There are domain-specific, often symbolic, constraints that exclude certain states as possible solutions. The cost can also include many complex factors and dependencies. To give some idea of the state-space size, consider the problem of placing 3 components on a 100 × 50 grid; each component can be placed on a square in this grid and no two components can occupy the same square. One possible state (i.e., a placement) is to place the components in squares given by the coordinates (10,10), (15,15) and (20,20). Clearly, the total number of possible placement for 3 components = 5000 × 4999 × 4998 = 124925010000 ≈ 125 billion; this is the size of the state-space.

It can be proved that many of these combinatorial optimization problems are intractable i.e., there are no known efficient systematic search algorithms that find the optimal solution for all input instances of the problem. Fortunately, if the optimal solution is too hard to find, a solution that is “sufficiently” close to it is often acceptable in practice. A common approach is then to define heuristic search algorithms that efficiently find such “near optimal” solutions most of the time. The heuristics refer to domain-specific knowledge or expertise to control and expedite the search. Among the many heuristic search algorithms, we shall outline one rather simple approach (and describe a C implementation skeleton for it) called hill climbing and use it to introduce some useful terms that describe common limitations of search algorithms. We then present a well-known probabilistic local

1 Girish is a scientist at the Tata Research Development and Design Centre (TRDDC), Pune, India, a division of Tata Consultancy Services. His research interests are in Artificial Intelligence and Mathematical Logic. He can be contacted at girishp@pune.tcs.co.in.
search technique called simulated annealing (SA) that mimics the natural process of slow cooling of liquids that leads to a solid that has the lowest energy. We provide a general C implementation of the SA algorithm and illustrate it by applying it to the problem of producing an aesthetic drawing of a given graph.

**Hill Climbing**

Hill climbing is a greedy nearest-neighbour approach to search the state-space to locate a lowest cost solution. It uses domain knowledge in the form of a “distance” function that returns an estimate of how close a given state is to the global minimum. This function is provided separately by the user for each problem domain e.g., the distance function for the placement problem is different from that for TSP. Figure 1 shows the C program skeleton of the hill climbing search algorithm, which is actually for a specific version called the steepest-ascent hill climbing or gradient search. The problem specific definitions of the C structure `STATE` and the assumed functions need to be provided for using this code.

It is easy to see the limitations of the hill-climbing algorithm for locating the global maximum. Hill-climbing often reaches a local minimum i.e., a state that has a lower cost than all its neighbours but which is not a global minimum. It may also reach a plateau i.e., a state whose neighbouring states all have the same cost and hence no next state can be chosen for further exploration. The hill-climbing algorithm can be modified to include simple strategies to deal with these problems; e.g., backtracking to some earlier state, taking a jump to some random state (e.g., when a plateau is encountered), moving in several directions simultaneously etc. Nevertheless, in general, hill-climbing has been found to be limited in practical combinatorial optimization problems. Simulated annealing (SA) is another local search algorithm (among a variety of other techniques) that overcomes some of the limitations of hill-climbing and often succeeds in efficiently locating good low-cost solutions.
// max. no. of neighbouring states for a given state
#define MAX_NEIGHBOUR 100

// external functions assumed (specific to each problem domain)
extern double Cost(STATE *pS); // returns the cost of given state
extern int Solution(STATE *pS); // 1 if given state is a solution; 0 otherwise
// generate neighbour states of given state; return no. of neighbour states
extern int GetNeighbours(STATE *pS, STATE Next[MAX_NEIGHBOUR]);

// hopefully find and return the lowest cost state
// STATE is a problem-specific representation of the structure of a state
// pS0 and pS are pointers to given initial state and low cost state found.
int HillClimbing(STATE *pS0, STATE *pS)
{
    STATE Next[MAX_NEIGHBOUR];
    int i, n, index;
    double c0, c, c1;

    if ( Solution(pS0) ) // found
    {
        CopyState(pS,pS0);
        return(1);
    }

    CopyState(pS, pS0); // initialize
    c0 = Cost(pS0); // cost of initial state
    while ( (n = GetNeighbours(pS,Next)) > 0 ) // get neighbours of s
    {
        index = 0; // index (in Next) of lowest cost neighbour
        c = Cost(&Next[0]);
        for(i = 0; i < n; i++) // do for each neighbour state
        {
            if ( Solution(&Next[i]) ) // found
            {
                CopyState(pS,&Next[i]);
                return(1);
            }
            if ( (c1 = Cost(&Next[i])) < c )
            {
                index = i;
                c = c1;
            }
        } // end for
        if ( c < c0 ) // found a lower cost neighbour
        {
            CopyState(pS, &Next[index]);
            c0 = c;
        } else // reached local maximum
        {
            break;
        } // end while
    }

    return(0); // global solution not found; S contains low cost state found
}

Figure 1. C skeleton for Steepest-Ascent Hill Climbing Algorithm

Simulated Annealing

It is well known in physics that when a hot liquid is cooled slowly (i.e., when annealed) it reaches a crystalline form. The atoms (or molecules) in a crystal are arranged in a regular and ordered
configuration; this state has the lowest (potential + kinetic) energy and consequently highest stability. On the other hand, when the hot liquid is cooled rapidly, the end result is an amorphous solid – this state has a somewhat higher energy. Qualitatively speaking, this happens because when the liquid is cooled slowly, the atoms have time to rearrange themselves till they achieve a state of lowest energy at that temperature – the so-called thermal equilibrium. In a thermal equilibrium at temperature T (in °K), the system achieves a state of energy E with a probability \( p(E) \equiv e^{-E/kT} \) where \( k = 1.3804 \times 10^{-16} \) erg per degree is a constant of nature, called the Boltzmann constant. An interesting thing to note about this Boltzmann distribution is that it allows the system to be in a higher energy state, even in a thermal equilibrium, although with a small chance. This chance decreases with temperature T, so that lower the temperature, higher the chance that the system is in the lowest energy state.

Metropolis and later Kirkpatrick et al realized the similarities between the annealing process and a general search for optimal solution in a combinatorial optimization problem. Essentially, the probability, at a temperature T, with which the search algorithm moves from a state s1 of energy E1 to another state s2 of energy E2 is given by \( e^{(E_1 - E_2)/kT} \). Clearly, whenever \( E_2 < E_1 \) the system will definitely move to s2 but when \( E_2 > E_1 \), the state change is probabilistic.

For applying search techniques to combinatorial optimization problems, we need to:

- Clearly define the information associated with each state.
- Identify an initial state, which is often generated randomly.
- Define a function that (often probabilistically) returns a state in the “neighbourhood” of a given state.
- Define a Boolean function that returns true if the given state is a solution (i.e., a minimum cost state) and false otherwise.

In addition, for applying SA to combinatorial optimization problems, we need to define analogues of temperature and cost. A common approach is to define energy as a function that returns the “cost” for a given state; the definition of the energy function includes all relevant factors that affect the cost. The Boltzmann constant \( k \) is often taken to be 1. The notion of temperature is often purely artificial (i.e., does not correspond to any real aspect of the problem) and is often defined to vary from a given maximum value \( T_{\text{max}} \) to a given minimum value \( T_{\text{min}} \). A number of cooling or annealing strategies are used to systematically drop the temperature from \( T_{\text{max}} \) till it reaches \( T_{\text{min}} \); the most common is the geometric annealing schedule where the next temperature is obtained from the current temperature by the geometric series formula \( T_{\text{next}} = \gamma \times T_{\text{current}} \), where \( \gamma \) is a given constant between 0 and 1. For instance, when \( T_{\text{max}} = 1000 \), \( T_{\text{min}} = 10 \) and \( \gamma = 0.5 \), the sequence of temperatures obtained using geometric annealing schedule will be 1000, 500, 250, 125, 62.5, 31.25, 15.625 and 7.8125 which is less than \( T_{\text{min}} \); at this point the schedule terminates. A high-level description of the SA algorithm is given below. The termination condition is often \( T < T_{\text{min}} \); but additional factors can be included here. Also, since the expression \( e^{(E_1 - E_2)/kT} \) need not be always between 0 and 1, it has to be scaled appropriately so that comparison with the output of a standard random number can be made.
Choose an initial state $s_0$ and an initial temperature $T_0$
$s = s_0; T = T_0$;
while (termination condition is not satisfied) do
    repeat the following a fixed number of times
        if (state $s$ is a solution) then return($s$);
        choose a new state $s'$ from the neighbourhood of $s$;
        if (state $s'$ is a solution) then return($s'$);
        $E :=$ energy($s$);
        $E' :=$ energy($s'$);
        if ($E' < E$ or random() < $e^{-\frac{(E' - E)}{kT}}$) then 
            $s := s'$; $E := E'$;
    end repeat;
    decrease the temperature $T$;
end while
return($s$); // whatever found so far

Figure 2. High-level description of the SA algorithm

A C implementation of the SA algorithm is shown below. Note that to apply this C implementation to a specific combinatorial problem, we need to provide some additional problem-specific details. We shall illustrate this process by applying this C implementation to a specific problem.

```c
/*
 * Module: Simulated Annealing
 * Programmer: Girish Keshav Paishikar
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "simann.h"  // problem-specific information needed by SA
#include "sim_app.h"  // problem-specific information needed by SA

// function prototypes
int NextTemp(TEMP *pt);
BOOL Boltzmann(ENERGY e1, ENERGY e2, TEMP t);
BOOL CoolEnough(TEMP t);

// problem-specific functions assumed
extern BOOL NeighbourState(STATE *pS, STATE *pNew);
extern BOOL IsSolution(STATE *pS); // TRUE if given state is an acceptable solution
extern ENERGY Energy(STATE *pS); // compute and return cost or energy of given state
extern int CopyState(STATE *pDest, STATE *pSrc);

// $s_0$ is the initial state and $t_0$ is the initial temperature
// Limit is the no. of attempts to explore neighbourhoods at a given temp.
BOOL SimAnneal(STATE *pInit, TEMP t0, int Limit, STATE *pDest)
{
    STATE s;   // current state
    TEMP t;    // current temp.
    ENERGY e, e_new; // energy of current and new state
    BOOL success = FALSE;
    int i;

    CopyState(&s, pInit); // make current state = initial state
    e = Energy(&s); // compute current energy
    t = t0;
```
C o p y S t a t e ( p D e s t ,  p I n i t ) ;   / /  c o p y  i n i t i a l  s t a t e  i n t o  p D e s t

w h i l e  (  ! C o o l E n o u g h ( t )  )
{
    i f  (  I s S o l u t i o n ( & s )  )  / /  d o n e
    {
        s u c c e s s  =  T R U E ;
        b r e a k ;
    }
    f o r  (  i = 0 ,  s u c c e s s  =  F A L S E ;  i < L i m i t ;  i ++ )
    {
        i f  (  ! N e i g h b o u r S t a t e ( & s ,  p D e s t )  )  / /  g e n e r a t e  n e w  s t a t e  i n  p D e s t
            b r e a k ;    / /  n o  m o r e  s t a t e s  i n  t h e  n e i g h b o u r h o o d
        i f  (  I s S o l u t i o n ( p D e s t )  )  / /  d o n e
        {
            s u c c e s s  =  T R U E ;
            b r e a k ;
        }
        e _ n e w  =  E n e r g y ( p D e s t ) ;
        p r i n t f ( " t e m p  =  % l f  i  =  % d  e  =  % l f  e _ n e w  =  % l f \n " ,  t ,  i ,  e ,  e _ n e w ) ;
        i f  (  B o l t z m a n n ( e ,  e _ n e w ,  t )  )  / /  m a k e  p D e s t  n e w  c u r r e n t  s t a t e  ?
        {
            C o p y S t a t e ( & s ,  p D e s t ) ;
            e  =  e _ n e w ;
        }
    }   / /  e n d  f o r
    N e x t T e m p ( & t ) ;   / /  g e t  t h e  n e x t  l o w e r  t e m p e r a t u r e
}  / /  e n d  w h i l e
C o p y S t a t e ( p D e s t ,  & s ) ;  / /  c o p y  l a s t  s t a t e  i n t o  p D e s t
r e t u r n ( s u c c e s s ) ;

B O O L  B o l t z m a n n ( E N E R G Y  e 1 ,  E N E R G Y  e 2 ,  T E M P  t )
{
    d o u b l e  x ;
    i f  (  e 2  <  e 1  )  r e t u r n ( T R U E ) ;
    x  =  e x p ( ( - ( e 2  -  e 1 ) )  /  t ) ;
    i f  (  x  < =  1 . 0  & &  r a n d ( )  <  x )  r e t u r n ( T R U E ) ;
    i f  (  r a n d ( )  <  ( x  -  f l o o r ( x ) ) )  / /  x  >  1 ,  s o  c h e c k  d e c i m a l  f r a c t i o n  o f  x
        r e t u r n ( T R U E ) ;
    r e t u r n ( F A L S E ) ;
}

/ /  i m p l e m e n t  c o o l i n g  s c h e d u l e ;  r e t u r n  t h e  n e x t  l o w e r  t e m p
i n t  N e x t T e m p ( T E M P  * p t )
{
    T E M P  t 1  =  ( * p t ) ;
    ( * p t )  =  t 1  *  T E M P _ F A C T O R ;
    r e t u r n ( 0 ) ;
}

/ /  t r u e  i f  g i v e n  t e m p  <  a  f i x e d  t h r e s h o l d ;  s t o p  c o n d i t i o n  f o r  S A
B O O L  C o o l E n o u g h ( T E M P  t )
{
    i f  (  t  <  M I N _ T E M P  )  r e t u r n ( T R U E ) ;
    r e t u r n ( F A L S E ) ;
}

/ *
*  M o d u l e :  S i m u l a t e d  A n n e a l i n g
*  F i l e :  s i m a n n . h
*  P r o g r a m m e r :  G i r i s h  K e s h a v  P a l s h i k a r
*/
#ifndef SIMANN_H
#define SIMANN_H

Aesthetic Graph Drawing using SA

A number of practical applications involving combinatorial optimization have used SA. We describe an interesting problem of automatically generating an aesthetic drawing of a given graph on an M × N grid (say, on screen), as discussed in [2]. The task here is to take a given graph of n vertices and e edges (representing connectivity between vertices) and place the n vertices on an M × N grid such that when the adjacent vertices are connected by straight line edges, the resulting graph looks aesthetically neat. Of course, there are many criteria for what constitutes an aesthetic drawing of a graph: vertices are evenly distributed in the grid, not too many are near the centre or near the borders, edge lengths are not too small nor too long, edge crossings are minimized and so on.

The additional C source code needed for applying the SA algorithm to the graph drawing problem is shown below. The C structure GRAPH contains the connectivity of the given graph. A state (as stored in the C structure STATE) represents a placement of all the n vertices of a given graph on the M × N grid i.e., a state associates a position, in the form of an x- and a y-coordinate, with each vertex in a given graph. The C function RandomState generates a random placement for all the vertices and is used to generate the initial state. The C function NeighbourState generates a neighbour state (for the given state) by randomly selecting a vertex and moving it by one square (up, down, right, left, up-right, up-left, down-right or down-left).

The function Energy computes the cost associated with a given placement of a given graph. Several factors contribute [2] to the calculation of the energy E(s) for a state s; we have implemented only the three, as given in the following formula for E(s):

\[
E(s) = \sum_{i,j=1}^{n} \frac{\lambda_i}{d_{ij}^2} + \sum_{i=1}^{n} \lambda_2 \left( \frac{1}{r_i^2} + \frac{1}{l_i^2} + \frac{1}{t_i^2} + \frac{1}{b_i^2} \right) + \sum_{k=1}^{e} \lambda_3 d_k^2
\]

Here, n and e are the number of vertices and edges in the given graph, d_{ij} is the Euclidean distance between vertex i and vertex j (in the given placement), r_i, l_i, t_i, b_i are the distances of the i^{th} vertex from the right, left, top and bottom borders respectively, d_k is the length of the k^{th} edge and \lambda_1, \lambda_2, \lambda_3 are the parameters used to control relative importance of the three factors. Contribution of the first factor to increases as the vertices move closer to each other and thus SA will try to prevent vertices from coming too close to each other. Contribution of the second factor to E(s) increases as a vertex moves closer to the borders of the M × N grid and thus SA will try to prevent vertices from coming too close to the borders. Contribution of the third factor to E(s) increases as lengths of edges increase and thus SA will try to prevent edges which are too long. Reference [2] suggests more
factors which contribute to $E(s)$ and which can be easily added. Increasing $\lambda_1$ with relative to other parameters causes SA to prefer pictures with smaller distances between vertices. Increasing $\lambda_2$ relative to $\lambda_1$ causes SA to push the vertices towards the centre and decreasing it causes SA to use more drawing space near the borders. Increasing $\lambda_3$ causes SA to prefer shorter edges. When either $d_i = 0$ or when either of $r$, $l$, $t$, $b$, or $h$ is 0, then we assume that the corresponding energy takes a large value denoted in the program by the constant MAX_ENERGY.

The function InitGraph fills up a global variable with the input graph that is to be placed. The function SimAnneal implements the SA algorithm. $T_{\text{max}}$ and $T_{\text{min}}$ are encoded in the constants MAX_TEMP and MIN_TEMP respectively.

As a small example of the application of the SA algorithm, here is a simple graph and its placements generated by the SA algorithm on a $30 \times 20$ grid. Note that the algorithm did not generate a hexagonal placement (possibly because it did not explicitly attempt to prevent edge crossings); however, the generated placement does look symmetrical in a different way. By adding more factors to the energy calculations and by selecting appropriate values for the parameters (e.g., for $\lambda_1$, $\lambda_2$ and $\lambda_3$), the SA algorithm can be used to generate pleasing placements for more complex graphs; see [2]. Even further complex factors that characterize aesthetic drawing of a graph can also be added; e.g., for balancing the area occupied by the convex hull of the graph and the remaining “blank” area.

![Graph](image)

Figure 4. A graph and a placement generated for it by the SA algorithm.
/ * Module: Graph Drawing using Simulated Annealing */
#define SIMULATED_ANNELING

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "simann.h"
#include "graph.h"
#include "g_draw.h"

// global variables
GRAPH G;
STATE S;

// Generate a random placement for the given graph
// return 0 upon completion
int RandomState(STATE *pS)
{
    int i, j, n, x, y;
    BOOL duplicate;

    n = pS->pG->nVertexCount; // no. of vertices in the graph
    for (i = 0; i < n; i++)
    {
        x = RANDOM(0, (MAX_X - 1));
        y = RANDOM(0, (MAX_Y - 1));

        // ensure that the position generated is not already allocated to
        // some other vertex
        for (j = 0, duplicate = FALSE; j < i; j++)
        {
            if (pS->aXPos[j] == x && pS->aYPos[j] == y)
            {
                duplicate = TRUE;
                break;
            }
        } // end for
        if (duplicate)
        {
            --i; // force retry for ith vertex
        }
        else
        {
            pS->aXPos[i] = x;
            pS->aYPos[i] = y;
        }
    } // end for

    pS->aXPos[n] = END_MARKER;
    pS->aYPos[n] = END_MARKER;
    return(0);
}

int PrintState(STATE *pS)
{
    int i, n;

    n = pS->pG->nVertexCount; // no. of vertices in the graph
    for (i = 0; i < n; i++)
    {
        printf("vertex i=%d: X = %d Y = %d\n", i, pS->aXPos[i], pS->aYPos[i]);
    } // end for
    return(0);
}
// get a random integer between 0 and Max (inclusive). Max must be > 0.
// return -1 on error
int GetRandom(int Max)
{
    if ( Max < 1 )
        return(-1);
    return( rand() % (Max+1) );
}

// return a state within the neighbourhood of given state
// Essentially, picks up a vertex randomly and moves it randomly to a
// neighbouring square (up, down, left, right, right-up, right-down,
// left-up, left-down)
BOOL NeighbourState(STATE *pS, STATE *pNew)
{
    int n, j, v_index, x, y, x1, y1, delta_x, delta_y, sign_x, sign_y;
    BOOL duplicate;
    n = pS->pG->nVertexCount;
    do  // till the position of selected vertex is successfully changed
        { 
            v_index = RANDOM(0, (n - 1));  // select vertex randomly
            x = pS->aXPos[v_index];  // current pos of the selected vertex
            y = pS->aYPos[v_index];

            delta_x = RANDOM(0,1);
            delta_y = RANDOM(0,1);
            sign_x = RANDOM(0,1);
            sign_y = RANDOM(0,1);

            if ( sign_x )  // increment x coordinate
                x1 = ( (x + delta_x >= MAX_X) ? (x) : (x + delta_x) );
            else  // decrement x coordinate
                x1 = ( (x - delta_x < 0) ? (x) : (x - delta_x) );

            if ( sign_y )  // increment y coordinate
                y1 = ( (y + delta_y >= MAX_Y) ? (y) : (y + delta_y) );
            else  // decrement y coordinate
                y1 = ( (y - delta_y < 0) ? (y) : (y - delta_y) );

            // ensure that the position generated is not already allocated to
            // some other vertex
            for ( j = 0, duplicate = FALSE; j < n; j++ )
                {
                    if ( pS->aXPos[j] == x1 && pS->aYPos[j] == y1 )
                        {  
                            duplicate = TRUE;
                            break;
                        }
                }
        }  // end for
    while ( ( duplicate || (x == x1 && y == y1) ) );  // do till position of vertex changes

    pNew->aXPos[v_index] = x1;  // report the new position
    pNew->aYPos[v_index] = y1;

    return(TRUE);
}

// TRUE if given state is an acceptable solution
BOOL IsSolution(STATE s)
{
    return(FALSE);  // no fixed final state for graph drawing
}

// compute and return cost or energy of given state
ENERGY Energy(STATE *pS)
{
    int i, j, n, x1, y1, x2, y2, v1, v2;
    ENERGY e = 0.0;
    double d, d1, d2, d3, d4;
    double c1, c2, c3;

n = pS->pG->nVertexCount; // no. of vertices in the graph
for (i = 0; i < n; i++)
{
    x1 = pS->aXPos[i]; // position of ith vertex in the graph
    y1 = pS->aYPos[i];
    v1 = pS->pG->aVertexID[i];

d1 = DIST_SQR(x1,y1,(MAX_X-1),y1); // right border
d2 = DIST_SQR(x1,y1,0,y1); // left border
d3 = DIST_SQR(x1,y1,x1,(MAX_Y-1)); // top border
d4 = DIST_SQR(x1,y1,x1,0); // bottom border
e = e + LAMBDA_2 * 
    ( (d1 > 0.0) ? (1 / d1) : (MAX_ENERGY) ) + // 0.0
    ( (d2 > 0.0) ? (1 / d2) : (MAX_ENERGY) ) + // 0.0
    ( (d3 > 0.0) ? (1 / d3) : (MAX_ENERGY) ) + // 0.0
    ( (d4 > 0.0) ? (1 / d4) : (MAX_ENERGY) ) ); // 0.0
for (j = 0; j < n; j++)
{
    x2 = pS->aXPos[j];
    y2 = pS->aYPos[j];
    v2 = pS->pG->aVertexID[j];

    // sqrt of distance between (x1,y1) and (x2,y2)
d = DIST_SQR(x1,y1,x2,y2);
    if ( j < i )
    {
        e = e + ( (d > 0.0) ? (LAMBDA_1 / d) : (MAX_ENERGY) );
    }
    if ( j < i && adjacent_v(pS->pG, v1, v2) ) // only once for an edge
    {
        e = e + LAMBDA_3 * d;
    }
} // end for
return(e);
}

int CopyState(STATE *pDest, STATE *pSrc)
{
    int i, n;
    n = pSrc->pG->nVertexCount; // no. of vertices in the graph
    pDest->pG = pSrc->pG;
    for ( i = 0; i <= n; i++ ) // copy END MARKER also
    {
        pDest->aXPos[i] = pSrc->aXPos[i];
        pDest->aYPos[i] = pSrc->aYPos[i];
    }
    return(0);
}

// return true if given vertices are adjacent in given graph
BOOL adjacent_v(GRAPH *pGraph, int V1, int V2)
{
    int i;
    for ( i = 0; i < pGraph->nEdgeCount; i++ )
    {
        if (((pGraph->aEdge[i].nVertexID1 == V1 &&
            pGraph->aEdge[i].nVertexID2 == V2)) ||
            ((pGraph->aEdge[i].nVertexID1 == V2 &&
            pGraph->aEdge[i].nVertexID2 == V1)) )
            return(TRUE);
    }
    return(FALSE);
main()
{
    STATE SFinal;
    initGraph(&G); // initialize G to contain K3,3
    S.pg = &G;
    RandomState(&S); // create a random layout for G
    SimAnneal(&S, MAX_TEMP, 600, &SFinal);
    PrintState(&SFinal);
}

// init the given graph to contain a hexagon
int InitGraph(GRAPH *pG)
{
    pG->nGraphID = 10;
    pG->eIsDirected = 0;
    pG->eIsDisconnected = 0;
    pG->eHasParallelEdge = 0;
    pG->nVertexCount = 6;
    pG->nEdgeCount = 6;
    pG->aVertexID[0] = 100;
    pG->aVertexID[1] = 101;
    pG->aEdge[0].nVertexID1 = 100; pG->aEdge[0].nVertexID2 = 101;
    pG->aEdge[1].nVertexID1 = 101; pG->aEdge[1].nVertexID2 = 102;
    pG->aEdge[2].nVertexID1 = 102; pG->aEdge[2].nVertexID2 = 103;
    pG->aEdge[4].nVertexID1 = 104; pG->aEdge[4].nVertexID2 = 105;
    pG->aEdge[5].nVertexID1 = 105; pG->aEdge[5].nVertexID2 = 100;
    pG->aEdge[6].nVertexID1 = END_MARKER; pG->aEdge[6].nVertexID2 = END_MARKER;
    return(0);
}

/* File: g_draw.h */
#define G_DRAW_H
#define MAX_X 30
#define MAX_Y 20
#define SQR(x) ((x) * (x))
#define DIST_SQR(x1,y1,x2,y2) ( SQR((x1)-(x2)) + SQR((y1)-(y2)) )
#define MAX_ENERGY 100000.0 // max energy (infinity)
#define LAMBDA_1 0.5
#define LAMBDA_2 0.001
#define LAMBDA_3 0.000001
typestruct {
    GRAPH *pg;
    int aXPos[MAX_VERTEX];
    int aYPos[MAX_VERTEX];
} STATE;

/* File: graph.h */
#undef GRAPH_H
#define GRAPH_H
#define END_MARKER -1 // terminator for lists
#define MAX_VERTEX 25
#define MAX_EDGE ((MAX_VERTEX * (MAX_VERTEX - 1)) / 2)
#define MAX_DEGREE (MAX_VERTEX - 1)
#define MAX_WIDTH (MAX_VERTEX - 1)
#define RANDOM(Min,Max) ((Min) + GetRandom((Max-Min)))
typestruct {
    int nVertexID1;
    int nVertexID2;
} EDGE;
Conclusions

A number of other heuristic algorithms have been invented for combinatorial optimization problems; e.g., AO* search, constraint satisfaction, means-end analysis, genetic algorithms etc. These algorithms yield acceptable “near optimal” solutions for a variety of problems. But there is no known universal algorithm that works efficiently for all problems. We examined a particular probabilistic heuristic optimization algorithm, called simulated annealing. SA mimics the natural process of slow cooling of liquids that leads to a solid form that has the lowest energy. We provided a C implementation of the general SA algorithm and applied it to the problem of producing an aesthetic drawing of a given graph, an application taken from [2].

A number of possible enhancements have been attempted for SA. The SA algorithm has been parallelized in various ways to obtain speedup benefits on multi-processor systems. Experimenting with different types of annealing schedules is another possible direction.

SA itself is not always suitable for all combinatorial optimization problems. Empirically it has been observed that for SA to work satisfactorily, the cost function should not contain narrow and steep valleys. The acceptable “near minimal” solutions should not be within such narrow and steep valleys. The energy (cost) function should change “smoothly” upon changes in states. It should also be efficiently computable, since it is invoked a large number of times. SA usually finds a near optimal solution but not the global minimum itself, if it has a low probability of being found. Finding suitable temperature schedule and best values of control factors for SA often needs careful experimentation. In spite of these problems, SA is being successfully used in a large number of practical problems, including VLSI chip design and layout, floor planning, flow shop scheduling, channel routing, graph drawing, image processing, coding theory, graph colouring and partitioning, satisfiability etc.

Acknowledgements. I gratefully acknowledge the support of Prof. Mathai Joseph. I also wish to thank Dr. Manasee Palshikar for her patience and strengths.
References


