Lecture 21:
Heuristic Methods (1997)

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

x *is* majority element *of a set* S *if the number of times it occurs is* $> |S|/2$*. Give an* $O(n)$ *algorithm to test whether an unsorted array* S *of* n *elements has a majority element.*

Sorting the list and checking the median element yields an $O(n \log n)$ algorithm – correct, but too slow.

Observe that if I delete two occurences of *different* elements from the set, I have not changed the majority element – since n is reduced by two while the count of the majority element is decreased by at most one.

Thus we can scan the set from left to right, and keep count of how many times we see the first element before we see an instance of a second element. We delete this pair and continue. If we are left with one element at the end, this is

the only candidate for the majority element.
We must verify that this candidate is in fact a majority element, but that can be tested by counting in a second $O(n)$ sweep over the data.

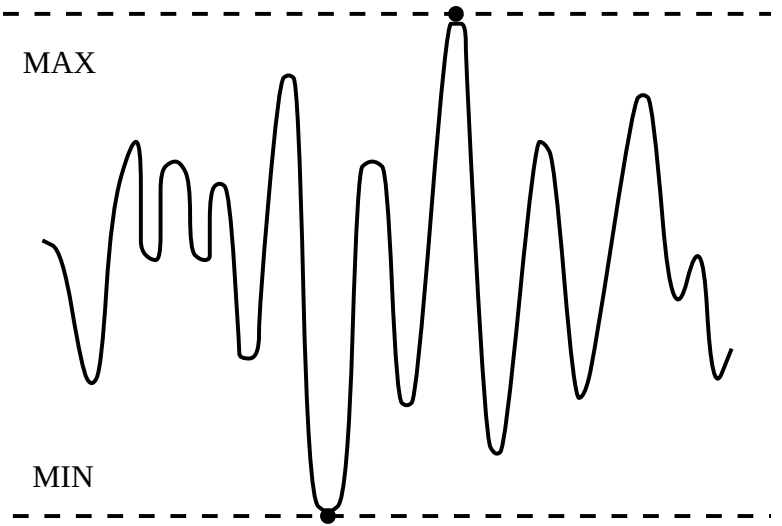
Combinatorial Optimization

In most of the algorithmic problems we study, we seek to find the best answer as quickly as possible.

Traditional algorithmic methods fail when (1) the problem is provably hard, or (2) the problem is not clean enough to lead to a nice formulation.

In most problems, there is a natural way to (1) construct possible solutions and (2) measure how good a *given* solution is, but it is not clear how to find the best solution short of searching all configurations.

Heuristic methods like simulated annealing give us a general approach to search for good solutions.


MAX
MIN

Simulated Annealing

The inspiration for simulated annealing comes from cooling molten materials down to solids. To end up with the globally lowest energy state you must cool slowly so things cool evenly.

In thermodynamic theory, the likelihood of a particular particle jumping to a *higher* energy state is given by:

$$e^{(E_i - E_j)/(k_B T)}$$

where E_i, E_j denote the before/after energy states, k_B is the Boltzman constant, and T is the temperature.

Since minimizing energy is a combinatorial optimization problem, we can mimic the physics for computing.

Simulated-Annealing()

Create initial solution S

Initialize temperature t

repeat

for $i = 1$ to *iteration-length* do

Generate a random transition from S to S_i

If $(C(S) \leq C(S_i))$ then $S = S_i$

else if $(e^{(C(S)-C(S_i))/(k \cdot t)} > random[0, 1))$

then $S = S_i$

Reduce temperature t

until (no change in $C(S)$)

Return S

Components of Simulated Annealing

There are three components to any simulated annealing algorithm for combinatorial search:

- *Concise problem representation* – The problem representation includes both a representation of the solution space and an appropriate and easily computable cost function $C(s)$ measuring the quality of a given solution.
- *Transition mechanism between solutions* – To move from one state to the next, we need a collection of simple transition mechanisms that slightly modify the current solution. Typical transition mechanisms include swapping the position of a pair of items or inserting/deleting a single item.

- *Cooling schedule* – These parameters govern how likely we are to accept a bad transition, which should decrease as a function of time. At the beginning of the search, we are eager to use randomness to explore the search space widely, so the probability of accepting a negative transition is high. The cooling schedule can be regulated by the following parameters:
  - *Initial system temperature* – Typically $t_1 = 1$.
  - *Temperature decrement function* – Typically $t_k = \alpha \cdot t_{k-1}$, where $0.8 \leq \alpha \leq 0.99$. This implies an exponential decay in the temperature, as opposed to a linear decay.
  - *Number of iterations between temperature change* –

Typically, 100 to 1,000 iterations might be permitted before lowering the temperature.

– *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 to accept a negative transition whenever

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

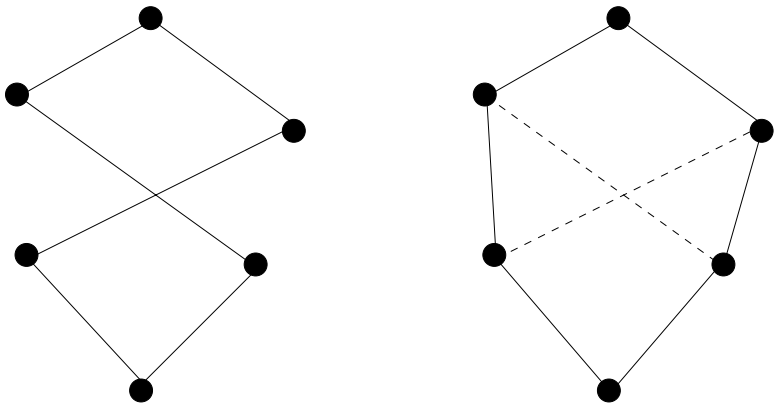
where r is a random number $0 \leq r < 1$. The constant c normalizes this cost function, so that almost all transitions are accepted at the starting temperature.

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

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

Traveling Salesman Problem

Solution space – set of all $(n-1)!$ circular permutations.
Cost function – sum up the costs of the edges defined by S.
Transition mechanism – The most obvious transition mechanism would be to swap the current tour positions of a random pair of vertices S_i and S_j. 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. Better would be to swap two edges on the tour with two others that replace it



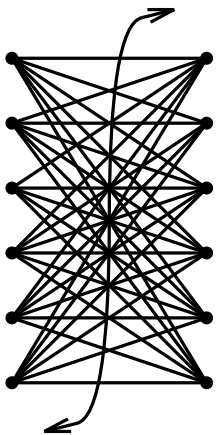
Since only four edges change in the tour, the transitions can be performed and evaluated faster. Faster transitions mean that we can evaluate more positions in the given amount of time.

In practice, problem-specific heuristics for TSP outperform simulated annealing, but the simulated annealing solution works admirably, considering it uses very little knowledge

about the problem.

Maximum Cut

Given a weighted graph, partition the vertices to maximize the weight of the edges cut.



This NP-complete problem arises in circuit design applications.

Solution space – set of all 2^{n-1} vertex partitions, represented

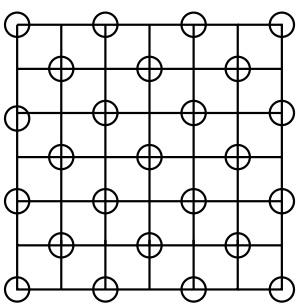
as a bit string.
Cost function – the weight of the edges which are cut.
Transition mechanism – move one vertex across the partition.

$$\Delta f = (\text{weight of old neighbors - weight of new neighbors})$$

This kind of procedure seems to be the right way to do maxcut in practice.

Independent Set

An independent set of a graph G is a subset of vertices S such that there is no edge with both endpoints in S. The maximum independent set of a graph is the largest such empty induced subgraph.

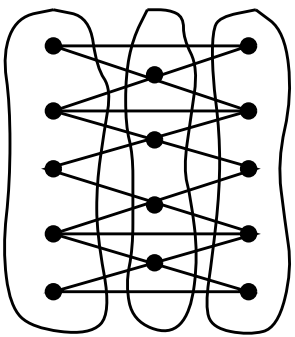


Solution space – set of all 2^n vertex subsets, represented as a bit string.

Cost function – $C(S) = |S| - \lambda \cdot m_S/T$, where λ is a constant, T is the temperature, and m_S is the number of edges in the subgraph induced by S.
The dependence of $C(S)$ on T ensures that the search will drive the edges out faster as the system cools.
Transition mechanism – move one vertex in/out of the subset.
More flexibility in the search space and quicker $\triangle f$ computations result from allowing non-empty graphs at the early stages of the cooling.

Chromatic Number

What is the smallest number of colors needed to color vertices such that no edge links two vertices of the same color?



The solution is complicated by the fact that many vertices have to shift (potentially) to reduce the chromatic number by one.

To insure that the proposed colorings are biased in favor of

low cardinality subsets (i.e. 28 red, 1 blue, and 1 green is better than 10 red, 10, blue, and 10 green), we will make certain colors more expensive than others.

By weighting the colors $w_{j+1} < 2w_j - w_1$ (ex: 100, 99, 97, 93, 85, 69, 37) we get faster convergence, although certain configurations might be cheaper than ones achieving the chromatic number! This can be enforced with a more complicated scheme.

By Brooks' Theorem, every graph can be colored with $\Delta + 1$ colors. In fact Δ colors suffice unless G is complete or an odd-cycle.

Solution space – all possible partitions of vertices into $\Delta + 1$ color classes, where Δ is the maximum vertex degree.

Cost function – $\Sigma_{i=1}^{\Delta+1} w_i(|V_i| - \lambda|E_i|)$, where $\lambda > 1$ is the

penalty constant.
Transition mechanism – randomly move one vertex to another subset.

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 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 an example of the kind of messy, multicriterion optimization problems for which simulated annealing is ideally suited.

We are given a collection of a rectangular modules $r_1, \ldots, r_n$, each with associated dimensions $h_i \times l_i$. For each pair of

modules r_i, r_j, we are given the number of wires w_{ij} that must connect the two modules. We seek a placement of the rectangles that minimizes area and wire-length, subject to the constraint that no two rectangles overlap each other.

Solution space – The positions of each rectangle. To provide a discrete representation, the rectangles can be restricted to lie on vertices of an integer grid.

Cost function – A natural cost function would be

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

where λ_{area}, λ_{wire}, and $\lambda_{overlap}$ are constants governing the impact of these components on the cost function.

Transition mechanism – moving one rectangle to a different location, or swapping the position of two rectangles.

Lessons from the Backtracking contest

- As predicted, the speed difference between the fastest programs and average program dwarfed the difference between a supercomputer and a microcomputer. Algorithms have a bigger impact on performance than hardware!
- Different algorithms perform differently on different data. Thus even hard problems may be tractable on the kind of data you might be interested in.
- None of the programs could efficiently handle all instances for $n \approx 30$. We will find out why after the midterm, when we discuss NP-completeness.

- Many of the fastest programs were very short and simple (KISS). My bet is that many of the enhancements students built into them actually showed them down! This is where profiling can come in handy.
- The fast programs were often recursive.

Winning Optimizations

- Finding a good initial solution via randomization or heuristic improvement helped by establishing a good upper bound, to constrict search.
- Using half the largest vertex degree as a lower bound similarly constricted search.
- Pruning a partial permutation the instant an edge was $\geq$ the target made the difference in going from (say) 8 to 18.
- Positioning the partial permutation vertices separated by b instead of 1 meant significantly earlier cutoffs, since any edge does the job.

- Mirror symmetry can only save a factor of 2, but perhaps more could follow from partitioning the vertices into equivalence classes by the same neighborhood.