Simulated Annealing and Parallel Resource Allocation

Arthur Sorkin
Lawrence Livermore National Laboratory

Prepared for publication in Parallel Computing

February 4, 1985
Abstract

In order to run a single problem on a parallel processor, the problem must be divided into a number of parallel resources, such as tasks, that can be allocated to processors. A number of conflicting constraints, such as communications costs and multiplexing costs, must be balanced in making the allocation. Allocating resources to processors in order to achieve optimum performance for the entire program is a difficult problem for which there are currently few heuristics.

There has been a great deal of recent interest in the Monte Carlo technique known as "simulated annealing" for the solution of optimization problems. Simulated annealing can be used to find near-optimal solutions of NP-complete problems in less than exponential time. Empirical results suggest that simulated annealing runs in polynomial time with a small exponent.

This paper discusses the application of simulated annealing to the parallel resource allocation problem. Some modifications to the basic annealing algorithm are suggested. Parallel algorithms for simulated annealing are also discussed.

Introduction

There is currently a great deal of interest in parallel processors. There are a number of research projects underway designing or using parallel computers, for example, S1, Mk. IIB [21], MMP [22], Cedar [14], and the Cosmic Cube [16-20]. A number of parallel computers are currently commercially available or will be so in the near future, for example, the Cray X/MP [19], HEP [27], and the Cray 2 [4]. Most of the commercial parallel processors have a small number of processors (no more than 16). However, the trend is towards massively parallel architectures with a large number of processors (i.e., hundreds or even thousands).

In order to run a single problem on a parallel processor, it is necessary to divide the problem up into pieces that can be assigned to different processors. The goal of this allocation is to achieve optimal (or
near optimal) performance for the whole problem. There are usually a number of conflicting constraints that must be balanced in doing the allocation. For example, when a number of tasks are allocated to a single processor, the processor must multiplex between them, thereby reducing its overall progress on each of them. If there are \( n \) tasks, each task will be executed at \((1/n)\)-th the speed that a single task would execute. This implies that to achieve optimal performance for the whole problem, each processor should be allocated as few tasks as possible. On the other hand, parallel tasks usually need to communicate with each other in the course of execution. Communication between two different processors is usually much more expensive than communication within a single processor. This implies that to achieve optimal performance for the whole problem, each processor should be allocated as many intercommunicating tasks as possible. Therefore, multiplexing costs and communications costs cannot both be optimized simultaneously and must be balanced instead. Depending on the problem, there may be many conflicting constraints that need to be balanced.

If \( p \) is the number of processors and \( n \) is the number of things to be allocated, then the number of possible allocations is \( p^n \). When there are only a few processors, it is feasible to do the allocation manually, but this rapidly becomes very difficult as the size of the problem or the number of processors increases. Even with as few as 16 processors, manually finding an optimal or near optimal allocation manually is not practical for any but very small problems.

There has been a great deal of recent interest in the technique known as simulated annealing for the solution of optimization problems with conflicting constraints [3]. Simulated annealing can be used to find near optimal solutions of problems that are exponential or even NP-complete in less than exponential time [1-3]. Empirical results suggest that simulated annealing runs in polynomial time with a small exponent [2].

**Representation of Allocation Problems**

Assume that we have a parallel processor with the processors connected according to some interconnection topology. The allocation of
resources to processors may be represented as a point in a multi-dimensional "configuration space" (also sometimes called phase space). Each dimension of configuration space represents the allocation of a single resource; there are as many orthogonal dimensions as there are resources to be allocated. In every dimension, there are as many points as there are processors. The topology of each dimension is the same as the interconnection topology of the processors. Figure 1a shows the one-dimensional configuration space representing the allocation of one resource to one of 10 processors. The dot shows that the single resource is allocated to processor number 4. Figure 1b shows a two-dimensional configuration space representing the allocation of two resources to one of 10 processors. In this case, the horizontal dimension represents the allocation of resource 1, and the vertical dimension represents the allocation of resource 2. The dot shows that resource 1 is allocated to processor 4 and resource 2 is allocated to processor 8. Figure 1c shows a one dimensional configuration space for a computer where the processors are connected in a ring. Figure 1d shows the two-dimensional toroidal configuration space representing the allocation of two resources for the same ring connected computer. Figure 1e shows a one dimensional configuration space where the processors are connected in a tree. In general, an n-dimensional configuration space can have an extremely complex topology.

Suppose that we have an n-dimensional configuration space, representing the allocation of n resources. With each point in that space, we may associate a cost or "energy" function, E, that numerically represents how good an allocation that configuration is. Larger E means a worse allocation, and smaller E means a better allocation. We can represent E with an (n+1)-st dimension orthogonal to the n-dimensional configuration space, so that the entire problem can be represented by a point in an (n+1)-dimensional space. Since each point in the n-dimensional configuration space has a value of E associated with it, the set of all E values forms an n-dimensional "energy surface" embedded in the (n+1)-dimensional space. A configuration space and its associated energy surface are shown in Figure 2.

With the above representation of the allocation problem, finding the optimal allocation may now be viewed as searching an energy surface
looking for the lowest point. One simple way to search is to do a random walk on the energy surface and accept all "level" or "downhill" moves (that result in equal or lower E values respectively) and reject all "uphill" moves (that result in a higher E value). A particular search corresponds to a path on the energy surface.

To do a simple search, start at a randomly chosen point. With the search at a particular point, randomly pick a direction to move and compare the value of E at the point one step away in the chosen direction with the value of E at the current point. Move to the new point if this is a level or downhill move; otherwise, stay at the current point and randomly choose a new direction to try. When the algorithm reaches a point where all adjacent points are uphill, then it halts.

**Simulated Annealing**

Unfortunately, there is a problem with this simple search algorithm. It will halt at a point that is a relative minimum, but that point may not be the absolute minimum. In fact, the simple search gets stuck at the bottom of the first "hole" that it enters. A better algorithm, called "simulated annealing" is described in [2]. Comparisons with known heuristics have shown that annealing is a valuable general-purpose optimization algorithm that is applicable to a large number of problems and that produces reasonable results [1,3]. Simulated annealing is better than a simple search because it randomly allows uphill moves. By making a random walk over the energy surface, but randomly allowing some uphill moves, simulated annealing doesn't get stuck in the first hole that it encounters.

Simulated annealing accepts all level and downhill moves. However, if the direction chosen by the algorithm is uphill, then a random number \( p \) between 0 and 1 is generated. This number is tested against \( e^{-\Delta E/T} \). If \( p \leq e^{-\Delta E/T} \) then the uphill move is accepted; otherwise the algorithm rejects the move and tries again from the current point. Here \( \Delta E \) is the difference between the E value at the new point and the E value at the current point. \( T \) ("temperature") is a control parameter. Note that \( \Delta E \) is guaranteed to be positive because this is an uphill move.

\( T \) is used to vary the probability that an uphill move will be accepted.
The value of $T$ is initially chosen large enough so that most uphill moves will be accepted. $T$ is then slowly lowered, causing the probability of accepting an uphill move to decrease; as the value of $e^{-\Delta E/T}$ approaches zero, the probability of accepting an uphill move approaches zero. Eventually, annealing halts at the bottom of a hole that it cannot climb out of.

The rate at which $T$ is lowered is called the "annealing schedule". If $T$ is lowered slowly enough, it has been shown that annealing will eventually find the absolute minimum point on the energy surface [1]. However, that annealing schedule is equivalent to doing an exponential search, which is not practical. Replacing $T$ with $0.9T$ every time the value of $T$ is changed results in more reasonable performance. For this annealing schedule, empirical tests indicate that the algorithm halts in polynomial time with a small exponent [1,2,3]. Other annealing schedules with reasonable running times are possible.

The value of $T$ is changed when the algorithm has actually changed position $n^*C_1$ times or when the algorithm has attempted to change position $n^*C_2$ times, whichever comes first. Here $n$ is the dimensionality of configuration space (which is the same as the number of resources to be allocated), and $C_1$ and $C_2$ are predefined constants such that $C_1$ is less than $C_2$. For example, $C_1$ might equal 10 and $C_2$ might equal 100. The $C_1$ criterion means that the algorithm has made a large enough number of moves at the current value of $T$. The $C_2$ criterion means that the algorithm has rejected a large number of moves at the current value of $T$, indicating that it is possibly becoming stuck in a hole. When $T$ changes value a predetermined number of times in a row (e.g., 3) because of the $C_2$ criterion, the algorithm halts.

**Modifications**

There are a number of problems with the algorithm as described in [2]. It is possible to have a hole on the energy surface with a "flat" bottom (i.e., a number of adjacent points have the same energy). Assume that the algorithm has reached the bottom of such a hole. Also assume that the value of $T$ is low enough so that there is a very small probability that the algorithm will make an uphill move. However, it can still make moves
between points on the bottom of the hole, since these configurations all have the same energy. In this situation, a large enough proportion of moves will be accepted that T will only change value because of the $C_1$ criterion. Since the $C_2$ criterion is never met, the algorithm will never halt!

For example, consider the flat-bottomed hole in Figure 3. Assume that the algorithm is currently at point B and that the probability of making an uphill move is extremely small (e.g. < .001). The algorithm will choose A or C as the next move with equal probability. If the choice is C, the move will be accepted because B and C have the same value of E. If the choice is A, 999 times out of 1000, the algorithm will reject the move and will try again from point B. Hence, almost half the moves from point B will be accepted. The situation at point C is identical to the situation at point B, so almost half the moves will be accepted at that point too. Hence, in a very large percent of the cases, almost half of the total moves will be accepted. In this case, this is sufficient to assure that T only changes value because of the $C_1$ criterion and never changes value because of the $C_2$ criterion.

One solution to this problem is to detect the fact that the energy is not changing from one move to the next. For example, modify the $C_1$ criterion to remember if the value of E has ever changed at the current T. If the algorithm is about to change the value of T because of the $C_1$ criterion and if E hasn't changed at this value of T, then this implies that we are at the bottom of a hole with a flat surface. We should count such a situation as potentially halting. Now, for either criterion, if there are three values of T in a row for which the system is potentially halting, then the algorithm halts.

Another problem with [2] is that it is possible for the system to halt in a configuration that is less optimal than one that was seen earlier in the search. The problem is that the algorithm has no memory. The solution is to modify the algorithm to remember the best configuration that it has ever seen. If the best configuration seen is not the halting configuration, then the best configuration should be used instead.

A third problem with [2] is the fixed, unit step size used during the random walk. In order to have a good search, the algorithm must, at least potentially, be able to reach every point in configuration space. If the
algorithm is restricted to unit moves, there are two possibilities. If the value of \( C_1 \) is at least as large as the size of a single dimension of configuration space, then every point in configuration space is reachable, but the search may be very time consuming. If the value of \( C_1 \) is less than that, then it is possible for the algorithm to tend to avoid parts of configuration space, giving a less than ideal search at each \( T \).

An obvious solution would be to allow arbitrarily long moves at every step. However, if each move can always be arbitrarily long, then at a low value of \( T \), the algorithm may halt prematurely. Assume that the algorithm has just entered a hole and that most points on the energy surface are higher than the one that has just been reached. Also assume that the size of the hole is very small compared to the size of configuration space. If all moves are equally likely, then almost all potential moves will be uphill and, therefore, will not be accepted. \( T \) will almost always change value because of the \( C_2 \) criterion, which can cause premature halting.

Some sort of mixed strategy is desirable. One solution is to have the maximum step size vary exponentially with \( T \) in much the same way as the probability of making an uphill move varies. Once the maximum step size is chosen for a move, then the move is made randomly to any legal configuration within that limit. For high values of \( T \), the probability will be high that the maximum steps will be large; for low \( T \) values, the probability will be high that the maximum step size will be one, with a smooth transition between the two. Thus, at high \( T \), every point in configuration space can potentially be reached in one step, without an extremely long search. But as the value of \( T \) decreases and the probability of making an uphill move approaches zero, the algorithm cannot halt prematurely.

This mixed strategy can be implemented as follows. Before each step is made, pick a random number, \( p \), between 0 and 1. For each possible value, starting with the diameter of a single dimension of configuration space (which is bounded above by the number of processors) and working down to 1, test to see if \( p \leq e^{\text{value}/T} \). If so, then choose that value as the maximum step size. If the test fails for all values greater than 1, then set the maximum step size to be 1. After the maximum step size is determined, the algorithm's next move is randomly chosen from the set of
all points up to that distance from the current point.

As was mentioned above, the initial value of $T$ must be picked so that the likelihood of accepting uphill moves is high. It was not stated in [2] how to choose this value. However, if $T(0)$ is chosen so that it equals the maximum value that $\Delta E$ can achieve in a single move, then $e^{-\Delta E/T} \approx 0.36$. Of course, not every move will have maximum $\Delta E$, so, in general, the value of $e^{-\Delta E/T}$ will be greater than 0.36, and a large percentage of the uphill moves will be accepted. Some percentage of uphill moves will be rejected, but, in practice, this choice of $T(0)$ seems to be sufficient.

Parallel Resource Allocation

An algorithm may be represented in a number of ways by graphs. For example, a data dependency graph depicts the flow of data between operations in a program. Data dependency graphs and their transformations have been extensively studied by Kuck, et al., for use in automatic vectorizing and parallelizing compilers [4-12]. Flow graphs are extensively used in optimizing compilers to represent the flow of control between parts of a program [13]. Petri nets have been used to represent parallel programs in the UCLA Graph Model of Computation [1]. Data-flow programs can also be represented by (possibly recursive) graphs [14-17].

As was noted above, parallel processors are connected according to some topology, for example, toroids and twisted toroids [26], hyper-cubes [24], shuffle-exchanges [23], banyon networks [25], and rings [21]. These interconnection topologies can also be represented by a graph.

The resource allocation problem can therefore be viewed as a graph mapping problem. The graph is used to represent the algorithm will be called the Problem Graph. The graph representing a parallel processor's topology will be called the Target Graph. Finding the optimal allocation of algorithm to processors is equivalent to finding the optimal mapping of the Problem Graph to the Target Graph. This problem can be solved with simulated annealing.

Assume that the Problem Graph is a data dependency graph. Each node represents an operation and each arc represents information that is communicated between the nodes. With each node, it is possible to associate a cost, e.g., relative execution time. With each arc, it is
possible to associate a cost, e.g., relative communications time. Assume that the Target Graph directly represents the processor interconnection topology. Each node represents a processor, and each arc represents a communication channel between processors. With each node and arc in the Target Graph, we can also associate costs, e.g., execution time and communication time respectively. In addition, we can also assign a cost to communications within a single node. Generally, communication within a single node are much less expensive than communications between nodes.

It is beyond the scope of this work to describe how the various cost values are arrived at. We will assume that such costs can be assigned by a very smart compiler [4-12]. Once such costs have been assigned, however, it is possible to use simulated annealing, as described above, to find a close-to-optimal mapping of the Problem Graph into the Target Graph. Note that a close-to-optimal mapping may map more than one Problem Graph node onto a Target Graph node and that some Target Graph nodes may have nothing mapped onto them.

As above, we will be doing a simulated annealing on an energy surface associated with configuration space. We have assumed that the various cost values have been given. In order to do simulated annealing, we must now define a cost (E) function for the problem as a whole.

When we map a number of operations onto a single processor, we must multiplex between them, effectively slowing the execution rate of all of them. We would like to minimize the number of operations per processor. We would also like the load to be balanced as much as possible among processors. Assume that one processor has many operations to multiplex among and that another processor that the first communicates with has very few operations to multiplex among. The first processor may not be able to proceed at full speed because it may need to wait for the second processor to generate data that it needs. These two criteria suggest that the total cost function should depend upon the load on all processors, while simultaneously minimizing the load on each individual processor. An obvious choice is the sum of the execution costs of the individual nodes. Unfortunately, this will not balance the load, since it is invariant under permutations of the operations among the processors. A non-linear function is required, and it was found that the sum of the squares of the execution costs works well for the total multiplexing cost.
In addition to multiplexing costs, the total cost function must also include total communications costs. We would like to minimize the cost of communication between pairs of Problem Graph nodes after they are mapped into the Target Graph. We would also like to balance the communications costs for reasons similar to the ones noted in the previous paragraph. One way to calculate communications costs is to look at nodes that are adjacent in the Problem Graph and see how far apart their mappings are in the Target Graph. Using the sum of the squares of the communications costs for all pairs of adjacent nodes in the Problem Graph works well.

The communications cost terms can be adjusted for the arc weightings in the Problem and Target Graphs. The multiplexing cost terms can be adjusted for the node weightings in the Problem and Target Graphs. The total cost function, $E$, is just the sum of the total multiplexing cost and the total communications cost.

A description of the annealing algorithm follows. It incorporates the improvements described above:

Problem Graph: $G$
Processor Graph: $H$
Energy Function: $E(m)$ (m is a mapping from $G$ to $H$)
Temperature Function:
$T(i)$: (initially $T(0)$)
- return $0.9 \times T(i-1)$;
end function;

Stepsizes Function:
stepsizes: (change randomly with every move and falls with increasing $i$)
- generate a random number $p$ in $[0,1]$;
- stepsizes = 1;
- loop: for $j =$ diameter($H$) to 1 by -1
  - if $p < e^{-j/T(i)}$ then
    stepsizes = $j$;

- 11-
exit loop;
endif;
- endloop;
- return stepsize;
- end function;

Constants: c1 - Average number of moves which must result in a new configuration in order for the temperature to change.
c2 - Average number of moves (whether they result in a new configuration or not) in order for the temperature to change.
k - Number of consecutive T values for which the algorithm must be potentially halting before it actually halts.

Algorithm A:

INITIALIZATIONS:

- randomly choose \( m : G \to H \)
- \( i := 0; \text{moves} := 0; \text{counter} := 0; \text{termcount} := 0; \text{Eflag} := 0; \text{BestE} := E(m); \text{BestM} := m \);

Loop:
Repeat until termcount\( \geq k \);
- randomly choose a vertex \( v \) in \( G \);
- randomly choose a vertex \( w \) in \( H \) such that \( \text{dist}(m(v), w) \leq \text{stepsize}(i) \);
- define the mapping \( m' = m \), except that \( m'(v) = w \);
- Calculate \( E(m') \);
- if \( E(m) = E(m') \) then
  \( m := m' \); \( \text{counter} := \text{counter} + 1 \);
  (see if we ever changed energy at this temperature)
  if \( E(m) > E(m') \) then \( \text{Eflag} := 1 \); endif;
else
  generate a random number \( p \) in \( [0,1] \);
  if \( p = \exp(-(E(m') - E(m))/\langle T(i) \rangle) \) then
    \( m := m' \); \( \text{counter} := \text{counter} + 1 \);


{in this case, we know the energy has changed this temperature)
Eflag := 1;
endif;
endif;
{remember the best configuration seen}
- if E(m) < BestE then
  BestE := E; BestM := m;
endif;
- moves := moves + 1;
- if counter /[|G|] >= c1 then
  i := i + 1; counter := 0; moves := 0;
  if Eflag = 0 then termcount := termcount + 1; else termcount := 0; endif;
  Eflag := 0;
- elseif moves /[|G|] >= c2 then
  i := i + 1; termcount := termcount + 1; counter := 0; moves := 0;
  eflag := 0;
endif;
end repeat;

- Output BestM and BestE;

Results

The annealing algorithm as described immediately above was applied to a number of Problem Graph and Target Graphs. Some typical allocations are displayed in Figures X-Y. The average multiplexing cost and the average communication cost are summarized in Figure.

As can be seen from the examples and the summary, simulated annealing is able to produce a close-to-optimal mapping of the Problem Graph onto the Target Graph.

Reallocation
Graphs are too static a representation for many programs. Some programs structure change shape during execution in a way that depends upon the input data. No single graph can properly capture the structure of such a program. For example, consider adaptive quadrature, where the number of iterations done in each interval depends upon the input data. The depth and fan out of the program graph also depends on the input data [1]. More dynamic finite structures such as recursive graphs are better models for this kind of behavior then simple graphs. A recursive graph can be thought of as an ordinary graph that can have special nodes that when executed, cause the graph to unfold into a bigger graph. Recursive graphs are used in data-flow languages and computers, for example, see [14-17].

Because of its relatively slow execution speed, simulated annealing is only practical for computing allocations prior to program execution. However, it is possible to use simulated annealing to pre-compute the reallocation of an unfolded graph, given that the original graph had already been allocated. An example of this is shown in Figure XX. In order to do reallocation, an additional term must be added to E. The additional term takes into account the cost of moving a resource once it has been allocated. For much the same reasons as stated previously, we would like this cost to be balanced among the processors. The additional term, therefore, should be a sum of squares, where each term is the distance that a resource that was already allocated has been moved by the reallocation, possibly times a weight factor. A resource that is not moved by the reallocation contributes zero to the sum.

It is possible to apply simulated annealing to recursive graphs in simple cases by statically allocating the initial graph and then doing a reallocation when a recursive node is executed. This may still be too slow to do at run time, but for simple recursive graphs with only a very small number of recursive nodes, it is possible to precompute the reallocations statically. The precomputed reallocations can then be stored with the program at run time and can be looked up when needed.

Statically precomputing reallocations is not intended to be a serious solution of the dynamic allocation problem, but is only intended to be an interim measure to allow greater flexibility than doing a single static allocation. How to handle dynamic allocation remains a difficult open question.
The reason simulated annealing cannot be used for dynamic allocation is that it is too slow. Even for static allocations, it would be desirable if annealing could be speeded up. One way to do this is to execute annealing in parallel.

Given $p$ processors, one way to make a parallel annealing algorithm is to divide configuration space into $p$ pieces, each $1/p$-th the size of the entire space and allocate one piece to each processor. Each processor searches its piece using annealing, and then the best of the $p$ results are used as the solution. Since the number of moves made at each temperature depends upon the maximum diameter of configuration space, it is best to try to make the division into pieces in such a way as to reduce this number.

Another way to introduce some parallelism into simulated annealing is to notice that if $E$ is a sum of terms, changing a the value of a single term doesn't change the value of $\Delta E$ computed because of a change in a different term, provided that $\Delta E$ is computed either entirely before the change in the first term or entirely after the change in the first term. For example, consider $E = T_1 + T_2$. If we change $T_1$ to $T_1'$, then $\Delta E = T_1' - T_1$, which is independent of the value of $T_2$, provided that the value of $T_2$ doesn't change during the $\Delta E$ computation and update of $T_1$. The $T_2$ term cancels out. Since the choice of $T_1$ was arbitrary here, this analysis applies to any terms of $E$ that are pairwise independent. If we let processors choose terms at random to update, then annealing can proceed in parallel. We must not permit two processors update their term at the same time. If any terms depend upon the same data, then the evaluation of those terms must be mutually exclusive too.

Another way to speed up annealing is to notice that what we really want to do is thoroughly search configuration space without getting stuck in a hole prematurely. When annealing rejects an uphill move, the search stays in the same location in configuration space as the algorithm randomly tries a new direction. An iteration of the algorithm where the location stays the same contributes nothing to the search. In terms of searching, it is a wasted move. It would be better if the algorithm had to
move to a new location at every iteration. One way to do this would be to randomly pick one of the $n$ dimensions to move in. Then throw weighted $m$-sided dice, where $m$ is the number of neighbors in the chosen dimension of the current point. For all real examples, the number of immediate neighbors is bounded by a small constant. The relative weight associated with a given neighbor is just equal to $e^{-\Delta E/T}$ for uphill moves and is equal to 1 for level or downhill moves. The weight associated with each neighbor is just the relative weight divided by the sum of all $m$ relative weights. Once again, varying $T$ varies the probability than an uphill move will be taken, relative to other possible moves from the same point. As $e^{-\Delta E/T}$ becomes smaller, the weights will more and more favor making level and downhill moves over uphill moves. When the average $e^{-\Delta E/T}$ becomes small enough, the algorithm will become "stuck" near the bottom of a hole. However, the algorithm can move away from a relative minimum no matter how small $e^{-\Delta E/T}$ is, because it must always make some move at every step even if all possible moves are uphill. But as $e^{-\Delta E/T}$ gets smaller and smaller, the weights will more and more strongly favor the algorithm moving level or downhill if that is a possibility, and therefore it will tend to move back towards the relative minimum. In the case of a flat bottomed hole, the algorithm will tend to remain on the flat bottom. When $T$ is small enough, the algorithm should either oscillate in a small neighborhood of the relative minimum or oscillate on the flat bottom. The algorithm halts when either of these conditions are detected.

For the parallel resource allocation problem, the algorithm is as follows.

Problem Graph: $G$
Processor Graph: $H$
Energy Function: $E(m)$  ($m$ is a mapping from $G$ to $H$
Temperature Function:
$T(i)$: (initially $T(0)$)
- return $0.9 \times T(i-1)$;
end function;
Constants:  
c1 - Average number of moves which must result in a new configuration in order for the temperature to change.

c2 - Euclidean diameter of a local neighborhood of a relative minimum used to detect potential halting.

k - Number of consecutive $T$ values for which the algorithm must be potentially halting before it actually halts.

Algorithm B:

INITIALIZATIONS:

- randomly choose $m: G \rightarrow H$
- $i := 0$; moves := 0; termcount := 0; BestE := $E(m)$; BestM := $m$

Loop:
Repeat until termcount$\geq k$;
- randomly choose a vertex $v$ in $G$;
  
  {calculate the relative weights}
- totweight := 0;
- Loop:
  - For all vertexes $w$ in $H$ such that $\text{dist}(m(v),w) \leq 1$;
  - define $m'_w := m$ for all points in $G$ except $v$ and for $v$, $m'_w(v) := w$;
  - Calculate $E(m'_w)$;
  - $\Delta E_w := E(m'_w) - E(m)$;
  - if $\Delta E_w \leq 0$ then relweight$_w := 1$;
    else relweight$_w := e^{-\Delta E_w / T}$; endif;
  - totweight := totweight + relweight;
endloop;

  {calculate the weights}
- Loop:
  - For all $w$;
  - weight$_w := \text{relweight}_w \cdot \text{totweight}$;
endloop;
- divide \([0,1]\) into disjoint intervals of length weight \(w\);
- generate a random number \(p \in [0,1]\);
- take \(m := m' \times w\), where \(w\) is the interval into which \(p\) fell;
  (detect the potential halting condition)
- if \(\Delta E_w \neq 0\) then \(\text{Eflag} := 1\);

(remember the best configuration ever seen)
- if \(E(m) < \text{BestE}\) then
  \(\text{BestE} := E; \text{BestM} := m\);
  endif;
- \(\text{moves} := \text{moves} + 1\);

- new configuration isn't in queue, then delete oldest element with lowest count and
  insert new configuration with \(\text{ref\_count} = 1\); else
  \(\text{ref\_count} := \text{ref\_count} + 1\) for configuration;

- If any configuration in queue has \(\text{ref\_count} > c1 \times n\) then
  \(\text{termcount} := \text{termcount} + 1\);

- If \(\text{Eflag} = 1\) then
  \(\text{termcount} := 0; \text{Eflag} := 0;\)
  else \(\text{termcount} := \text{termcount} + 1; \text{Eflag} := 0;\) endif;

(see if \(T\) should be changed)
- if \(\text{counter} / |G| \geq c1\) then
  \(\{\text{change } T\}\)
  \(i := i + 1;\)
  \(\text{moves} := 0;\)
  \(\text{termcount} := 0;\)
  endif;

end repeat;

- Output \(\text{BestM}\) and \(\text{BestE}\);  

- 18-
Acknowledgement

The author would like to acknowledge the very helpful comments of Fran Berman and Howard Davidson. This work was performed by Lawrence Livermore National Laboratory under U.S. Department of Energy Contract W-7405-ENG-48.

References


15 S. Skledzielewski and J. Glauert, "IF1 - An Intermediate Form for Applicative Languages," Draft 9, University of California, Lawrence


21 H. Davidson, S-1 project, Lawrence Livermore National Laboratory, personal communication.


UCLA Graph Model & Petri Nets

Cray X/MP and Cray 2