Lattice Model Protein Simulations:
Phase Transitions and Rugged Energy Landscapes

Katherine Schwarz
BioEngineering 290E
May 20, 2003
1. Project Goals

1.1. Objectives

This project is a Monte Carlo Markov chain simulation of the thermodynamics and kinetics of a lattice model protein. The goals are to use simulated annealing to find the minimum-energy, or “native” state, and to find whether our crude model of the interaction energy can give rise to a cooperative phase transition characterized by a latent heat. This report first describes an object-oriented design for Monte Carlo simulation of a lattice model protein, and then describes some programs that use the design.

1.2. The model

For ease of programming, the simplest possible lattice model was used: a protein consists of a string of beads on a cubic lattice. The beads are either H (hydrophobic) or P (polar), representing broad classes of amino acid side chains. The potential energy of interaction is only between non-bonded nearest neighbors and takes the values +1 for unlike residues and –1 for like residues. This is a very crude way of representing the physical interactions between side chains in an aqueous environment, which favors clustering together of hydrophobic residues.

Two sequences were provided to us:

Sequence 1: HHPH PPHH HPPP PPHH PHPH HHHH PPPP HPHP PHPP

and

Sequence 2: HHHH PHHP HHHH PHHP PHHP HPHP HPHP PHPP PHHP PPPP

2. Object-oriented design

Object-oriented programming was used for this project because it is a good way to write modular, reusable code. Code was written in C++, compiled with g++, and run on Socrates, a Sun Fire v880 Server that provides general Unix computing services for the Berkeley campus. Socrates has four 900 MHz UltraSPARC-III processors and eight gigabytes of memory, and runs the Solaris 2.8 operating system. When the load on Socrates was high, programs were also run on C199, a 2 (900 Mhz) processor SunFire 280R running Solaris 8, and Cory, a 4 (450 Mhz) processor UltraSparc E450 running Solaris 8.

2.1. Objects

The key objects designed for the project were:

- A Lattice, which has a collection of LatticeNodes
- A Chain, which has a linked list of Beads
- A MonteCarloSimulator, which makes moves on a Chain

The main programs that used these objects could then be written at a high level of abstraction and were quite short.
2.1.1. Lattice

A LatticeNode needs to know whether it is occupied, so that beads can know whether they are allowed to move into it, and what its neighboring nodes are. Since the energy function depends only on nearest neighbors, the energy of a bead at a node can be quickly computed just by examining the neighbors of the node, without needing to check the location of every bead in the chain. Therefore, the space lattice is represented as a collection of LatticeNode objects. Each LatticeNode has a list of its neighbor nodes, each associated with a direction, and can report the interaction energy of either an H or a P bead placed into it. (This is the simplest possible version of the cell list method.) With this method the energy of a trial move can be computed in constant time, independent of the length of the chain.

A Lattice object contains a three-dimensional array of nodes. Its only function is to construct itself, by telling each node who its neighbor nodes are. The Lattice is twice the length of the chain in size, to allow the chain to start at the center and proceed in any direction; this requires the initialization of a large amount of memory, so the Lattice object should persist for the entire program.
2.1.2. Chain
The Chain and Bead classes contain the bulk of the code. The Bead is the building block, and does most of the actual work.

Bead
A Bead knows its type (H or P), the lattice node where it is located, pointers to the previous and next beads on the chain, and the direction to the next bead. It is capable of computing the energy of its interactions with its neighbors, moving itself to different lattice nodes, and most importantly, it can propose a local trial move of itself and calculate the associated change in energy. When a bead is requested to propose a move,

1. It selects a local move, possibly a choice from a set of candidates.
2. If the move would result in a collision, it returns “false”, meaning the Monte Carlo simulation must reject the move. This enforces the exclusion of atoms from each other’s volume; it is equivalent to counting the energy of any conformation with two beads in the same node as minus infinity.
3. If no one- or two-bead move could be found because of the conformation of the chain, it also returns “false”.
4. Otherwise, the move and the associated change in energy is calculated and stored, so that the Monte Carlo simulator can accept or reject it by the Metropolis criterion.

The method used to select a local move is:

1. An end bead selects any position adjacent to the next bead at random, excluding its current position. If that position is already occupied, the move is rejected.

2. An interior bead first checks whether the chain is straight at its position (that is, the directions to the previous and next beads are parallel). If so, there is no local move available, and the move must be rejected.

3. An interior bead at a 90° angle may be able to make a corner or crankshaft move, but not both, since the bead is in a crankshaft (at the positions labeled “bead 1” or “bead 2” in the figure below) if and only if the opposite corner is already occupied, so that a corner move would be impossible anyway. Therefore, without loss of generality, the bead first considers a corner move. If the opposite corner is unoccupied, the corner move is proposed.
4. If the opposite corner is occupied, the directions of links between beads are examined to see whether this is bead 1 of a crankshaft. If it is, one of the three possible alternate orientations is selected at random. If the destination nodes are unoccupied, the crankshaft move is proposed; otherwise, the move must be rejected.

5. If this bead is not bead 1 of a crankshaft, it may be bead 2; if so, a candidate move is selected as before.

6. Finally, if the bead cannot make a corner move and is not in a crankshaft, the move must be rejected.

The bead does not actually perform the trial move until the Monte Carlo simulator requests it.

A bead can also compute its current contribution to the chain energy by finding the interaction energy with all the neighbors of its position, and then subtracting out the interactions with the previous and next beads linked along the chain.
**Bead**

- contents
- pointers to previous and next beads in chain
- location in lattice (pointer to LatticeNode)
- direction of link to next bead
- type of local move most recently proposed
- energy change associated with the local move

- move to new location
- compute current energy
- generate trial move (corner or crankshaft)
- perform trial move

**UML diagram for the Bead class.** FirstBead and LastBead are subclasses which inherit from it.

**Chain**

A Chain administers its linked list of Beads by reading in a sequence, creating a Lattice for it to occupy, and initializing the configuration. The initial configuration may be read from a file, may be a straight chain, or may be constructed by Rosenbluth chain growth at a given temperature. Rosenbluth growth was used in most cases as a way of generating a random configuration with a good probability of having an energy in the right range. (Growth in the forward or backward direction was selected at random with equal probability.) The probability of generating a given configuration by Rosenbluth growth is not the same as the Boltzmann factor of the configuration, but this effect can be washed out by allowing the Monte Carlo simulation to “settle”—make moves without recording data—for some number of moves in order for the distribution to approach the desired Boltzmann distribution.
When a Chain is requested to make a trial move, it simply picks one of its Beads at random with equal probability, and requests that bead to generate a trial move. The result (success or failure at generating a move) is then passed back to the Monte Carlo simulator which requested the move.

**UML diagram for the Chain class.**

### 2.1.3. Monte Carlo Simulator

The MonteCarloSimulator object is given an initialized Chain and a temperature to run at. It keeps a histogram of how many times each energy level has been visited. It needs only a few basic functions:

1. The move function asks the Chain to propose a local move. If no local move could be generated, the move is rejected, and the configuration remains the same for another time step. Since all local moves are reversible and the reverse move has the same probability as the original, the Metropolis criterion can be used:

   \[
   \text{probability of acceptance} = \min[1, e^{-\beta \Delta E}]
   \]

   where \( \Delta E = \text{proposed energy} - \text{current energy} \). This criterion ensures that the Boltzmann distribution of configurations will be maintained. If accepted, the Chain is then asked to make the move.

2. The run function loops for a given number of moves, occasionally reporting statistics on average energy and acceptance rate. After each move the histogram is updated with the current energy. Also, a record is kept of the minimum-energy
configuration seen so far. After running for the specified number of moves, the heat capacity is also reported.

3. The settle function is the same as the run function, except that records of energy are not kept. This is meant to allow the initial configuration to equilibrate at the given temperature.

4. The Simulator is also able to reset its temperature while keeping the chain configuration, for simulated annealing.

**Derived Class: NativeSeeker**

A NativeSeeker object is a type of MonteCarloSimulator that runs only until it reaches a given goal in energy, and then stops and reports the number of moves required. This simulator is used for simulated annealing and for finding the first passage time.

### 2.2. Optimizations

Since the Monte Carlo simulation must try a very large number of moves, several tricks were used to speed up the code that depend on the simple nearest-neighbor energy function.

1. Since there are only two types of bead and the interaction energy takes values of only +1 or −1, a LatticeNode does not actually need to examine each of its neighbors to determine the interaction energy of a bead placed there; it only needs to keep a count of the net number of neighbors, with each P counting as +1 and each H as −1. Then the interaction energy is equal to the net count for an H in the node, and the negative of it for a P; for example, if 3 neighbors are P and one is H, then the interaction energy would be −2 for a P placed in the node, and +2 for an H. Keeping this count means that a node can very quickly report the energy of a trial move that places a bead into it, although actually making the move is slowed down since all the surrounding nodes must update their counts. This optimization assumes that trial moves will be tested much more often than they are accepted.

2. Computing exponentials for the Metropolis criterion is computationally expensive. Since there are only a small set of possible energy changes, all of which are integers, all necessary Boltzmann factors can be computed in advance and stored in a table.

With this implementation, a MonteCarloSimulator was able to perform 1 million moves in about 2 seconds on a 900 MHz processor. (The code was compiled without optimization flags; they did not seem to make much difference.)

### 2.3. Advantages and Disadvantages

Some advantages of the design are:
• Object-oriented designs are meant to be reusable, with changes confined to as small a set of objects as possible. For example, it would be fairly easy to alter the Lattice object to use a different lattice such as the diamond lattice. It would only need to change how it connects nodes to each other; each node would now have 4 nearest neighbors instead of 6. The Chain and MonteCarloSimulator objects would not need any significant changes.

• With the cell list method, the time to compute energy of a trial move is constant, independent of the length of the chain. This would make it easy to run the program on longer chains (although the number of moves necessary to find an acceptable one would still grow with the size of the chain).

And some disadvantages are:

• Memory cost: The lattice needs to have enough nodes for the chain to start at the center and stretch out straight in any direction, plus a one node thick buffer zone of edge nodes, which are labeled as “occupied” so that the chain can’t enter the edge, where it might fall off. For a chain of maximum length 36, this means a $74 \times 74 \times 74$ lattice, with 405,224 nodes, must be created and kept in memory; this required 22Mb of memory while the program was running, and the requirement would increase as $N^3$ for a chain of length $N$, which could quickly become prohibitive. (The program might not really need this much space in the lattice, since most of the configurations sampled will be compact. If the typical radius of gyration is proportional to $N^{1/3}$, then the space required will only be proportional to $N$. However, I didn’t try any smaller sizes.)

• Optimizations used to speed up the code depend on the very simple nearest-neighbor energy function. A more realistic energy function would require a lot of redesign and would not run as fast.

3. Applications

3.1. Simulated Annealing

The process of simulated annealing mimics the physical process of annealing, where a molten system is cooled very slowly in order to reach a global minimum energy configuration. It is used to search for the global minimum in systems with many local minima separated by high barriers. The basic idea is that at the initial high temperature, the system is able to cross the barriers and sample the configuration space ergodically (that is, without being trapped in any subset of the space). Therefore, after a sufficient number of Monte Carlo moves, the system is most likely to be found in the basin of attraction of the deepest minimum. The temperature is then gradually lowered as the Monte Carlo simulation continues, so that the system is more and more confined in configuration space and eventually settles into a minimum. Sometimes it will settle into the wrong basin, so the simulated annealing process must be repeated many times.
The algorithm for simulated annealing is:

**Algorithm: Simulated Annealing**

Start at some initial temperature, high enough that there is a good probability of surmounting barriers, but not so high that there is no preference for lower energies; $kT=1.0$ was used.

Choose an initial configuration; Rosenbluth chain growth at the initial temperature was used.

Until a minimum-energy goal is reached or a maximum number of stages is exceeded:

- Run a Monte Carlo simulation, using local moves and Metropolis acceptance criterion as described above, for a fixed number of moves, $N_T$.

- Lower the temperature. A geometric cooling schedule was used, where the temperature is multiplied by a parameter $\alpha$ after each stage. This allows more time to be spent at colder temperatures, where barriers are harder to cross.

The main program `simanneal.cpp` (see appendix) implements this algorithm. It takes the sequence, $\alpha$, goal, $N_T$, and maximum stages as command-line parameters. The program simply creates a NativeSeeker object with the given goal, and runs it for $N_T$ steps at each temperature stage until the NativeSeeker reports that the goal is reached.

After some trial and error, $N_T = 10$ million and $\alpha=0.99$ seemed to work well and were used in a series of simulated annealing runs on the two sequences. The number of stages was limited to 100 (corresponding to a minimum temperature of $kT=0.37$) because if the system has settled into the wrong basin at this cold temperature, it will not be able to escape in any practical amount of time. This is demonstrated by the figure below, which plots the configuration energy, sampled at intervals of 500,000 steps, during the simulated annealing run for six typical runs with Sequence 1. As the temperature cools, the average energy decreases and the spread of energy values narrows. Three of the runs reach the goal of $-36$ and stop. The other three have clearly become trapped in basins of local minima $-33$ or $-34$ and are no longer sampling a wide enough range of energies to cross the barrier into the global minimum. This plot suggests that the height of the energy barriers is about $-10$, since when the top of the energies sampled during the run drops below this level, the trajectories are trapped.
The Monte Carlo process also becomes less and less efficient as the temperature cools, as shown by the decrease in acceptance rate during this relatively long run:
Because of the low acceptance rate, continuing the annealing for much longer would be fruitless.

35 out of 50 simulated annealing runs on sequence 1 (70%) succeeded in reaching the goal of -36 in energy. A histogram of the number of temperature stages completed is shown below:

All of the successful runs ended in the same configuration, so we can be confident the minimum-energy state is unique. It is maximally compact, completely filling a 3-by-3-by-4 block. Two views are shown below, with H beads as light colored spheres and P beads as dark ones:

Some notes on this configuration:

- There are 38 favorable contacts and 2 unfavorable ones for a net energy of –36.
- H and P beads are segregated from each other, although there is no distinction between core and boundary areas because the chain is not big enough.
- It is tightly bound: all possible moves out of this state cost at least +2 in energy.
- There are few pathways out of or into this state: the only possible moves are end moves and 5 crankshaft moves.
- The two antiparallel hairpins formed by the first 12 beads are suggestive of a beta sheet, although in reality beta sheets are formed by hydrogen bonds on the peptide backbone and not by side chain interactions.

All 75 of 75 simulated annealing runs on sequence 2 succeeded in reaching the goal of -36 in energy. The runs did not all reach the same configuration. Three basic configurations were found; that is, there are three different “native” states. (Also, each of the three allows small variations in the last beads on the chain.) This sequence is easier to fold because there are more allowable targets, and also because there are more kinetic pathways into the global minima.

39 of 75 runs found Configuration A, which is maximally compact:

- This configuration also has 38 favorable interactions and 2 unfavorable ones.
- The P bead at the end is “loose”: it has one favorable and one unfavorable interaction, so it can be moved out of the block at no cost. This means this configuration is actually a set of 3 equivalent ones.
- There are a lot more pathways (possible crankshaft moves) into or out of this state than for sequence 1. This is one reason why it is easier to fold.

23 of 75 runs found Configuration B, which fits into a 3-by-4-by-4 block:
This configuration has 36 favorable interactions and no unfavorable ones.
There are at least 3 different ways of arranging the last 3 P beads on the chain with the same energy, so this configuration is also a set of 3 equivalent ones.

13 of 75 runs found Configuration C, which fits into a 3-by-3-by-5 block:

This configuration also has 36 favorable interactions and no unfavorable ones.
There are also different ways of arranging the last 2 P beads on the chain with the same energy, so this configuration is also a set of equivalent ones.

**Faster Quenching**
If the temperature in simulated annealing is cooled too quickly, the system will be quenched: it will find a local minimum quickly without exploring the full range of configurations. For comparison, some simulated annealing runs were performed with $\alpha=0.90$ for faster cooling. With this schedule, none of 10 runs with sequence 1 succeeded at finding the global minimum, and only 15 of 20 runs with sequence 2 succeeded.

The number of possible configurations increases very rapidly with energy, while the Boltzmann factor decreases very rapidly; the population distribution over the energy states is the product of these two factors, producing a bell-shaped curve whose location gradually shifts with temperature. Also, after recording the number of times each energy level is visited, the heat capacity may be calculated by

$$C_V = \frac{\langle E^2 \rangle - \langle E \rangle^2}{(kT)^2}$$

That is, the heat capacity depends on the spread of energy levels visited. In a first order phase transition, we expect the heat capacity to exhibit a sharp peak, representing the absorption of significant latent heat with little temperature rise during the transition.

However, there is a problem with finding the distribution at low temperatures: starting from a random initial configuration, the system will be quasiergodic, unable to escape a local minimum as the Boltzmann factor for barrier crossing in the Metropolis algorithm becomes very small. The histogram reported by the simulator will represent only a small region around some local minimum, not the true equilibrium distribution. For example, in this series of runs, a smoothly varying bell-shaped curve was found for kT=1.0 through 0.5, but at kT=0.45, the system became stuck in a rather high local minimum and did not generate the correct distribution.

![Sequence #1 Population Distribution at Various T](image)
There are advanced methods for overcoming this problem, such as multiple histograms and parallel tempering, but they are beyond the scope of this report. Instead, the native state was given as the initial condition, ensuring that it would always be found. This means we are really looking at the melting process, rather than the folding process.

**Sequence 1 Melting**

Population distributions and heat capacity were examined by the main program `heatcap.cpp`, which starts a Monte Carlo simulation in the minimum energy configuration, allows it to settle for 10 million moves, and then collects data for 100 million moves. For temperatures above the critical temperature of about 0.5, the resulting distributions were essentially the same as when starting from a random configuration, but for lower temperatures, the population remains almost entirely in the global minimum.

The highest temperature where any population remains in the global minimum is \( kT=0.55 \); above this temperature, the simulation escapes into higher energy regions and does not return. The distribution at this temperature is:
Energy Counts Energy Counts Energy Counts
-36 2524462 -24 8538919 -12 37618
-35 0 -23 6948761 -11 19277
-34 347148 -22 5429048 -10 8012
-33 465570 -21 3964253 -9 3642
-32 1929817 -20 2766380 -8 2394
-31 3473775 -19 1840261 -7 975
-30 6592599 -18 1187667 -6 524
-29 8885177 -17 730056 -5 153
-28 10885259 -16 445781 -4 61
-27 11432009 -15 253047 -3 59
-26 11032757 -14 142250 -2 47
-25 10035485 -13 76740 -1 12
-24 0 0
(Counts are out of a total of 100 million.)

The fraction remaining in the global minimum exhibits a sharp decrease at about kT=0.5: (this plot also shows results from kT=0.51 and 0.52):

The peak in heat capacity coincides with the temperature where the fraction folded drops steeply. We can conclude that the model does indeed represent a cooperative phase transition. It is also physically reasonable that no states exist with energy of –35: there is an energy gap between the phases.

(The asymmetry of the heat capacity curve may be because of the quasi-ergodicity: the system is confined to the region of the global minimum and does not actually achieve the Boltzmann distribution.)

The melting temperature can be defined as the temperature where the folded and unfolded states are in equilibrium at equal concentrations; that is, the fraction in the
minimum energy state is 0.5. By interpolating between the points for kT=0.5 and 0.51, this temperature is 0.505873336 for sequence 1.

**Sequence 2 Melting**

All three minimum-energy configurations were used as the initial condition, but the results were similar for all of them. The population distribution generated by starting in configuration A is:

![Population distribution of sequence 2A](image)

Note that the distinction between frozen and unfrozen distributions is much less sharp than for sequence 1; it looks more like the bell curves gradually march to the left and bump up against the minimum possible energy, instead of suddenly concentrating in the minimum, as the temperature drops.

The fraction remaining in the minimum also decreases much more gradually as the temperature rises:
Note that the peak in heat capacity is not at the point where the frozen and unfrozen populations are equal. This does not appear to be a cooperative phase transition, and there is no energy gap. In this case it makes more sense to define the folding temperature as the location of the peak in heat capacity, because the population distributions at lower temperatures are somewhat suspect because of the quasi-ergodicity. The folding temperature occurs where the system is sampling the greatest possible range of energy levels (scaled by \((kT)^2\)), which is therefore the point of greatest heat capacity.

### 3.3. First Passage Time

Finally, we can examine the kinetics of the folding process, since the Monte Carlo move set consists of physically possible moves of the chain. The time to fold at the folding temperature was examined with the main program `firstpassage.cpp`, which initializes a chain by Rosenbluth growth, allows it to settle for a selected number of moves, and then runs a NativeSeeker simulation until the minimum energy state is reached or a maximum number of steps is exceeded.

#### Sequence 1 Kinetics

For sequence 1 the number of settling moves was 10 million and the maximum number of steps was 600 million; if the sequence has not folded in 600 million moves, the program gives up. Only 48 of 100 trials were able to achieve the folded state within this time limit, which suggests that the mean first passage time is much longer than 600 million moves, but running longer simulations would take too much computation time.
If we assume that the first passage time $t$ has an exponential probability distribution,

$$p(t) = \lambda e^{-\lambda t}, \quad \langle t \rangle = 1/\lambda$$

then the parameter $\lambda$ may be estimated from the fraction of times that exceed a maximum:

$$p(t > t_{\text{max}}) = \int_{t_{\text{max}}}^{\infty} \lambda e^{-\lambda t} \, dt = e^{-\lambda t_{\text{max}}}$$

Therefore, using $p(t > t_{\text{max}}) = 0.52$, the first passage time can be estimated by

$$\hat{\lambda} = \frac{-\ln p(t > t_{\text{max}})}{t_{\text{max}}}, \quad \langle t \rangle = 1/\hat{\lambda} = 918 \text{ million moves}$$

**Sequence 2 Kinetics**

For sequence 2 the number of settling moves was 10 million and the maximum number of steps was 100 million. 99 of 100 trials were able to fold within this time limit. A histogram of the times also shows a roughly exponential decay:
Since 99% of the sequences did fold within the time limit, we can just take the mean observed time, 19.2 million moves, as the mean first passage time.

Sequence 2 folds much more quickly than sequence 1 because it has more targets available and more kinetic pathways into them.

4. Conclusions

4.1. Properties of the sequences

This simulation showed that Sequence 1 does have a cooperative phase transition with a very steep melting curve coinciding with a very sharp peak in heat capacity. Sequence 2 can fold to any of three minimum energy configurations, and its melting curve is much more gradual. This is because Sequence 2’s minima are more accessible with more paths into them.

The simulations became quasiergodic below the melting temperature, failing to explore the entire configuration space and achieve the Boltzmann distribution. This demonstrates that the lattice model proteins tend to have rugged energy landscapes with many local minima separated by steep barriers, and often are frustrated (i.e., stick in a local minimum) when trying to fold. By examining the simulated annealing trajectories we can estimate that the typical height of barriers is about $E = -10$ (that is, the simulation must climb up to this level to cross the barrier).

Sequence 2’s landscape is evidently less rugged than Sequence 1, and (based on the success of 100% of the simulated annealing runs) the basins of attraction of the three minima must take up practically all the configuration space.
Sequence 1 has a unique folded state, which is typical of naturally occurring proteins. Sequence 2 is not representative of naturally occurring proteins; since protein function depends on fold, a protein that folded randomly to various different shapes would be useless, and natural selection would eliminate it.

Yue et al. tested some attempts to design lattice proteins to fit a given fold, and demonstrated that they tend to have a very large number of equivalent global energy minima; our sequences were presumably very carefully selected to have little or no degeneracy.

4.2. Possible improvements

This lattice model is too crude to really represent a peptide chain; for example, it has no way to represent hydrogen bonds along the backbone, which determine secondary structure, and the varying size of the side chains, which is crucial in the packing of the hydrophobic core. Also, the backbone of a real protein bends at angles much less sharp than 90°. The obvious next steps would be to choose a different lattice such as the diamond lattice or 210 lattice, and to use a much more detailed interaction energy function with more classes of beads, for example the BLN model.

The lattice model protein is also a good candidate for parallel tempering or multiple-histogram studies, which could overcome the quasi-ergodicity problem and give more reliable results at temperatures below the melting point.

Crude as the lattice model is, it provides a proof of principle: it is possible to predict a phase transition by simulation.
5. References


Appendices

Code:

**Basic Modules**

- `lattice.h`, `lattice.cpp`: define LatticeNode and Lattice classes
- `chain.h`, `chain.cpp`: define Bead and Chain classes
- `montecarlo.h`, `montecarlo.cpp`: define MonteCarloSimulator and NativeSeeker classes

**Programs using modules**

- `simanneal.cpp`: Simulated annealing, seeking a given minimum energy
- `heatcap.cpp`: Runs Monte Carlo simulation at a fixed temperature to examine population distribution and heat capacity
- `firstpassage.cpp`: Starts with a random configuration and runs until the folded state is found, reporting the number of steps required.

**Configurations:**

coordinates of the minimum energy states
// lattice.h
#define _LATTICE_H_

#include <string>
#include <iostream>

using namespace std;

const int N_NEIGHBORS = 6;
enum Direction {XPLUS, XMINUS, YPLUS, YMINUS, ZPLUS, ZMINUS, INVALID};

const string direction_name[] =
{ "Right", "Left-", "Front", "Back-", "Up---", "Down-", "Invalid"};

bool is_perpendicular(Direction a, Direction b);
bool is_antiparallel(Direction a, Direction b);
Direction opposite(Direction a);

const Direction CROSSPRODUCT[N_NEIGHBORS][N_NEIGHBORS] =
{ { INVALID, INVALID, ZPLUS, ZMINUS, YMINUS, YPLUS },
  { INVALID, INVALID, ZPLUS, ZMINUS, YPLUS, YMINUS },
  { ZMINUS, ZPLUS, INVALID, INVALID, XPLUS, XMINUS },
  { ZPLUS, ZMINUS, INVALID, INVALID, XMINUS, XPLUS },
  { YPLUS, YMINUS, XMINUS, XPLUS, INVALID, INVALID },
  { YMINUS, YPLUS, XPLUS, XMINUS, INVALID, INVALID }};

// Used to rotate any configuration to a standard orientation where the
// first displacement is in the +X direction.
// ROTATE[first][dir] gives the direction that dir should be rotated
to,
//   if first is the first displacement.
// e.g. ROTATE[YPLUS][dir] rotates Y+ to X+, so all directions are
//      rotated -90 degrees around the z axis.
const Direction ROTATE[N_NEIGHBORS][N_NEIGHBORS] =
{ { XPLUS, XMINUS, YPLUS, YMINUS, ZPLUS, ZMINUS },   // identity
  x+ => x+
  { XMINUS, XPLUS, YMINUS, YPLUS, ZPLUS, ZMINUS },   // 180 rotation
  x- => x+
  { YMINUS, YPLUS, XPLUS, XMINUS, ZPLUS, ZMINUS },   // -90 rotation
  y+ => x+
  { YPLUS, YMINUS, XMINUS, XPLUS, ZPLUS, ZMINUS },   // +90 rotation
  y- => x+
  { ZMINUS, ZPLUS, YPLUS, YMINUS, XPLUS, XMINUS },   // +90 rotation
  z+ => x+
  { ZPLUS, ZMINUS, YPLUS, YMINUS, XMINUS, XPLUS } };  // -90 rotation
z- => x+

// default constants
const int MAXLENGTH = 36;
const int MAXLATTICE = 2*MAXLENGTH + 2;
// A LatticeNode knows its neighbors and the interaction energy that
// an H or P at this node would have with them.

class LatticeNode {
public:
    LatticeNode();
    virtual void clear();

    // accessors
    bool is_occupied() const;
    bool is_empty() const;
    LatticeNode* neighbor(int i) const;
    bool is_neighbor(LatticeNode* other) const;

    int interaction_E(bool content_type) const;

    // mutators
    void add_neighbor(LatticeNode* new_neighbor, Direction d);
    void add_occ_neighbor(bool neighbor_type);
    void remove_occ_neighbor(bool neighbor_type);
    void occupy(bool content_type);
    void unoccupy(bool content_type);

protected:
    bool occupied;
    LatticeNode* neighbors[N_NEIGHBORS]; // pointers to all adjacent nodes
    int tot_neighbors; // net occupied neighbors, +P -H

    // birth control for unwanted functions
private:
    LatticeNode(const LatticeNode& rhs);
    LatticeNode& operator=(const LatticeNode& rhs);
};

// CLASS LatticeEdgeNode inherits from LatticeNode

class LatticeEdgeNode : public LatticeNode {
public:
    LatticeEdgeNode::LatticeEdgeNode();
    virtual void clear();
};

// CLASS Lattice

// A Lattice constructs a three-dimensional array of connected nodes
class Lattice {
public:
    Lattice(int s);
    ~Lattice();
    void clear();

    LatticeNode* centernode() const;
    LatticeNode* quarternode() const;

    bool on_edge(int x, int y, int z) const;

private:
    int size;
    int center;
    int quarter;

    LatticeNode* nodes[MAXLATTICE][MAXLATTICE][MAXLATTICE];
    LatticeNode* my_centernode;
    LatticeNode* my_quarternode;

    // prevent automatic generation of unwanted functions!!! don't define!!
    Lattice();
    Lattice(const Lattice& rhs);
    Lattice& operator=(const Lattice& rhs);
};

#if 0
#endif
// lattice.cpp

#include "lattice.h"

/////////////////////////////////////////////////////////////////////
//        Implementation of direction functions
//  Assumes directions are numbered 0, 1, 2, 3, 4, 5
bool is_perpendicular(Direction a, Direction b) {
  return (a/2 != b/2);
}

bool is_antiparallel(Direction a, Direction b) {
  return ((a/2 == b/2) && (a != b));
}

Direction opposite(Direction a) {
  return ( (a%2 == 0) ? (Direction)(a+1) : (Direction)(a-1) );
}

/////////////////////////////////////////////////////////////////////
//            CLASS LatticeNode implementation
LatticeNode::LatticeNode() :
  occupied(false), tot_neighbors(0)
{}  // note: neighbors[] contains garbage until neighbors are linked

// clear contents of nodes, while leaving coords and interconnections
void LatticeNode::clear() {
  occupied = false;
  tot_neighbors = 0;
}

bool LatticeNode::is_occupied() const {
  return occupied;
}

bool LatticeNode::is_empty() const {
  return (!occupied);
}

LatticeNode* LatticeNode::neighbor(int i) const {
  return neighbors[i];
}

bool LatticeNode::is_neighbor(LatticeNode* other) const {
  for (int i=0; i<N_NEIGHBORS; i++) {
    if (other == neighbors[i]) return true;
  }
  return false;
}
// Energy of interaction with spatial neighbors
// -1 for neighbors w/same polarity as occupant, +1 for opposite polarity
int LatticeNode::interaction_E(bool content_type) const {
    return (content_type ? -tot_neighbors : tot_neighbors);
}

// Connect to neighbor node in the given direction
void LatticeNode::add_neighbor(LatticeNode* new_neighbor, Direction d) {
    neighbors[d] = new_neighbor;
}

// Make a note whenever someone moves in next door; record net total
void LatticeNode::add_occ_neighbor(bool neighbor_type) {
    if (neighbor_type) tot_neighbors++;
    else tot_neighbors--;
}

// remove neighbor, adjust net total
void LatticeNode::remove_occ_neighbor(bool neighbor_type) {
    if (neighbor_type) tot_neighbors--;
    else tot_neighbors++;
}

// a chain link is placed at this node
void LatticeNode::occupy(bool content_type) {
    occupied = true;
    // inform my neighbors I'm here
    for (int i = 0; i < N_NEIGHBORS; i++) {
        neighbors[i] -> add_occ_neighbor(content_type);
    }
}

// vacate this node
void LatticeNode::unoccupy(bool content_type) {
    occupied = false;
    // inform my neighbors that content_type has been removed
    for (int i = 0; i < N_NEIGHBORS; i++) {
        neighbors[i] -> remove_occ_neighbor(content_type);
    }
}

// CLASS LatticeEdgeNode
// is always "occupied" to prevent chain from entering it
LatticeEdgeNode::LatticeEdgeNode()
void LatticeEdgeNode::clear() {
    tot_neighbors = 0;
}

///////////////////////////////////////////////////////////////////////
///
// CLASS LATTICE

// construct lattice, connect each node to neighbors
Lattice::Lattice(int s) :
    size(s), center((size-2)/2), quarter(size/4 + 1)
{
    for (int x = 0; x < size; x++) {
        for (int y = 0; y < size; y++) {
            for (int z = 0; z < size; z++) {
                nodes[x][y][z] = ( on_edge(x,y,z) ? new LatticeEdgeNode() : new LatticeNode() );
                // connect neighbors:
                if (x > 0) {
                    nodes[x][y][z]->add_neighbor(nodes[x-1][y][z], XMINUS);
                    nodes[x-1][y][z]->add_neighbor(nodes[x][y][z], XPLUS);
                }
                if (y > 0) {
                    nodes[x][y][z]->add_neighbor(nodes[x][y-1][z], YMINUS);
                    nodes[x][y-1][z]->add_neighbor(nodes[x][y][z], YPLUS);
                }
                if (z > 0) {
                    nodes[x][y][z]->add_neighbor(nodes[x][y][z-1], ZMINUS);
                    nodes[x][y][z-1]->add_neighbor(nodes[x][y][z], ZPLUS);
                }
            }
        }
    }
    my_centernode = nodes[center][center][center];
    my_quarternode = nodes[quarter][center][center];
}

// be careful to destroy all that was allocated with new, otherwise it leaks
Lattice::~Lattice() {
    for (int x = 0; x < size; x++) {
        for (int y = 0; y < size; y++) {
            }
for (int z = 0; z < size; z++) {
    delete nodes[x][y][z];
}
}
}

// Clear contents of lattice, but leave neighbor connections
// (EdgeNodes remain "occupied" to block the chain from writhing off the edge)
void Lattice::clear() {
    for (int x = 0; x < size; x++) {
        for (int y = 0; y < size; y++) {
            for (int z = 0; z < size; z++) {
                nodes[x][y][z]->clear();
            }
        }
    }
}

LatticeNode* Lattice::centernode() const {
    return my_centernode;
}

LatticeNode* Lattice::quarternode() const {
    return my_quarternode;
}

bool Lattice::on_edge(int x, int y, int z) const {
    return ((x == 0) || (y == 0) || (z == 0) ||
            (x == size-1) || (y == size-1) || (z == size-1));
}
// chain.h defines Class Bead and Class Chain
// Depends on lattice.h

#ifndef _CHAIN_H_
#define _CHAIN_H_

#include "lattice.h"

#include <iostream>
#include <string>
#include <cstring>   // old-fashioned C strings
using namespace std;

////////////////////////////////////////////////////////////////////
// CLASS Bead
// A Bead has contents, a location, and direction to next link
// It can determine its possible local moves, and do a move in given
// direction.
// It belongs to a doubly linked list.

enum LocalMoveType { END, CORNER, CRANKSHAFT1, CRANKSHAFT2, FAILEDMOVE }
const string move_name[] = { "end", "corner", "crankshaft", "crankshaft" }

class Bead {
friend class Chain;
friend class FirstBead;
friend class LastBead;

public:
    Bead(int i, bool newcontents, Bead* newprev);
    Bead(bool newcontents);
    void clear();

    LatticeNode* neighbor(Direction i) const;
    bool is_neighbor(Bead* other) const;
    virtual int current_E() const;
    void occupy(LatticeNode* where);
    virtual void moveto(Direction in, Direction out, LatticeNode* where);

    // overridden in First, Last

    // move generation: overridden in First and Last beads
    virtual bool generate_move();
    bool generate_corner();
    bool generate_crankshaft();
    inline bool is_crankshaft1() const;
    inline bool is_crankshaft2() const;
    bool crankshaft(LocalMoveType movetype,
                    Bead* piv1, Bead* c1, Bead* c2, Bead* piv2);
};
int deltaE() const;

void print_move(ostream& out = cout) const;
virtual void do_move();

protected:
const int my_index;       // position in chain; never change after created
const bool contents;      // never change after created
Bead *prev, *next;

LatticeNode* location;
Direction linkfrom;

LocalMoveType movetype;
Direction movedir;
int delta_E;     // energy change due to move

private:
Bead();
Bead(const Bead& rhs);
Bead& operator=(const Bead& rhs);

};

class FirstBead : public Bead {
public:
FirstBead(bool newcontents);
virtual int current_E() const;
virtual void moveto(Direction out, LatticeNode* where);

virtual bool generate_move();
virtual void do_move();

private:
FirstBead();
FirstBead(const FirstBead& rhs);
FirstBead& operator=(const FirstBead& rhs);

};

class LastBead : public Bead {
public:
LastBead(int i, bool newcontents, Bead* newprev);
virtual int current_E() const;
virtual void moveto(Direction in, LatticeNode* where);

virtual bool generate_move();
virtual void do_move();

private:
LastBead(bool contents);

};
CLASS Chain
A Chain has a collection of Beads with a given HP sequence.
Chain objects will last for the whole program.
A Chain constructs a space lattice where it will do its
cconfigurations
(large and expensive, so keep re-using the Chain)

class Chain {
public:
   // construction
   Chain(const char* target);
   ~Chain();
   void seq_from_string(const char* source);
   void place_straight();
   void write_config(ostream& out);
   void read_config(istream& in);
   void copy_config_from(Direction store[]);
   void copy_config_to(Direction store[]);
   void calc_energy();
   void clear();
   void print(ostream& out = cout) const;
   void print_config(const Direction config[], ostream& out = cout) const;
   // accessors
   bool is_empty() const;
   int energy() const;
   const char* getstring() const;
   int getlength() const;
   // mutators
   void initiate(bool res1, bool res2);
   void extend(int pos, Direction dir);
   bool generate_move();
   int deltaE() const;
   void print_move(ostream& out = cout) const;
   void do_move();
   // Rosenbluth chain growth
   double Rosenbluth_extend(int index, bool forward);
   double Rosenbluth_grow(double temperature);
private:
   char seqstring[MAXLENGTH];
   bool sequence[MAXLENGTH];
   const int seqlength;
   Bead* beads[MAXLENGTH];
   Lattice lattice; // all points in space must know their neighbors
int movelink;  // which link is moved
int chain_energy;

double expmbeta;   // used for Rosenbluth chain growth

// prevent the compiler from creating unwanted default functions
Chain();
Chain(const Chain& rhs);
Chain& operator=(const Chain& rhs);

};

int pick(double weights[], int n, double totweight);

#endif
// chain.cpp

#include "chain.h"           // which in turn reads lattice.h and
   localmove.h

#include <iostream>
#include <string.h>
#include <cstdlib>           // for drand48()
#include <cmath>
using namespace std;

/////////////////////////////////////////////////////////
//            CLASS Bead implementation
/////////////////////////////////////////////////////////
// New beads are constructed with no location and no link out
Bead::Bead(int i, bool newcontents, Bead* newprev)
   : my_index(i), contents(newcontents), prev(newprev), next(0),
      location(0), linkfrom(INVALID), movedir(INVALID)
{
   prev->next = this;
}

// use this only with FirstBead
Bead::Bead(bool newcontents) :
   my_index(0), contents(newcontents), prev(0), next(0),
      location(0), linkfrom(INVALID), movedir(INVALID)
{}

// Clear configuration (location and directions), but keep contents
// and position in chain
void Bead::clear() {
   location = 0;
   linkfrom = INVALID;
}

// neighbors of my spatial position
LatticeNode* Bead::neighbor(Direction i) const {
   return location->neighbor(i);
}

bool Bead::is_neighbor(Bead* other) const {
   return location->is_neighbor(other->location);
}

// Count interactions with spatial neighbors, but not with adjacent
// along the chain
int Bead::current_E() const {
   int energy = Location->interaction_E(contents);
}
// take away interactions with next and previous links
if (contents == next->contents) energy++;
else energy--;
if (contents == prev->contents) energy++;
else energy--;

return energy;
}

// put Bead at location. Used when extending chain.
// Caller will also have to update the links between Beads.
void Bead::occupy(LatticeNode* where) {
    location = where;
    location->occupy(contents);
}

// move Bead to new location. Used when making local moves.
void Bead::moveto(Direction in, Direction out, LatticeNode* where) {
    location->unoccupy(contents);
    location = where;
    location->occupy(contents);
    prev->linkfrom = in;
    linkfrom = out;
}

// determine local moves; overridden in First and Last
// Generate a local trial move. Return true if move was generated.
// Store the direction and type of move.
// Each type of move fails if it would create a collision.
bool Bead::generate_move() {
    movetype = FAILEDMOVE;
    movedir = INVALID;
    delta_E = 0;

    // If chain is straight here, can't make any local move.
    if (prev->linkfrom == linkfrom) return false;

    // Note that at most one type of move is possible: if it is able to
    // make a corner move, it is unable to make a crankshaft, and vice
    // versa.
    if (generate_corner()) return true;
    else return generate_crankshaft();
}

// Try to generate a corner move. Return true if move was generated.
// FAILURE (reject move) if there is a collision.
// Store the direction and type of move.
//        x     o
//              ^    destination: x is neighbor of bead i-1
//              |                 in direction of link from i
//        o----->o i
bool Bead::generate_corner() {
    // No need to specify direction, only one is possible
    // Check for collision. Return failure if collision
    if (prev->neighbor(linkfrom)->is_occupied()) return false;

    // Success!! We'll need delta E
    movetype = CORNER;
    // both locations' energy contain contributions from adjacent beads
    // on chain; cancelled out by subtracting
    delta_E = prev->neighbor(linkfrom)->interaction_E(contents) -
              location->interaction_E(contents);
    return true;
}

bool Bead::generate_crankshaft() {
    if (is_crankshaft1()) {
        return crankshaft(CRANKSHAFT1, prev, this, next, next->next);
    } else if (is_crankshaft2()) {
        return crankshaft(CRANKSHAFT2, prev->prev, prev, this, next);
    } else return false;
}

bool Bead::is_crankshaft1() const {
    return ((next->next != 0) && prev->is_neighbor(next->next));
}

bool Bead::is_crankshaft2() const {
    return ((prev->prev != 0) && prev->prev->is_neighbor(next));
}
// Randomly choose an alternate orientation of the crankshaft defined by
//   c1  c2
//   o-->o
//   ^   |       link from piv1 perpendicular to link from c1
//   |   v       link from piv1 antiparallel to link from c2
//   o   o
// piv1 piv2 : these are the pivots
// // FAILURE (reject move) if it would collide.
bool Bead::crankshaft(LocalMoveType movetype, Bead* piv1, Bead* c1, Bead* c2, Bead* piv2) {
    const int CRANK_MOVES = 3;  // possible rotations of crankshaft
    switch((int)(drand48()*CRANK_MOVES)) {
      case 0:
        movedir = c2->linkfrom;  // this turns crankshaft 180 degrees
        break;
      case 1:
        movedir = CROSSPRODUCT[c1->linkfrom][piv1->linkfrom];  //90deg rt hand
        break;
      case 2:
        movedir = CROSSPRODUCT[piv1->linkfrom][c1->linkfrom];  //90deg lft hand
        break;
      default:
        cerr << "bad random number in generate_crankshaft\n";
        exit(2);
    }
    // Check for collision; return failure if collision
    if (piv1->neighbor(movedir)->is_occupied() ||
        piv2->neighbor(movedir)->is_occupied() ) return false;
    // Success!!  We'll need delta E.
    this->movetype = movetype;
    int newenergy = piv1->neighbor(movedir)->interaction_E(c1->
                    contents) +
                    piv2->neighbor(movedir)->interaction_E(c2->contents);
    int oldenergy = c1->location->interaction_E(c1->contents) +
                    c2->location->interaction_E(c2->contents);
    // oldenergy contains interactions of c1 and c2 w/each other;
    // correct for those
    if (c1->contents == c2->contents) oldenergy += 2;
    else oldenergy -= 2;
    // oldenergy and newenergy both also contain energy of interaction
    // with the two pivots, but that's cancelled by subtracting them
    delta_E = newenergy - oldenergy;
    return true;
}
int Bead::deltaE() const {
    return delta_E;
}

void Bead::print_move(ostream& out) const {
    out << "Bead " << my_index << " proposes " << move_name[movetype] << " move toward " << direction_name[moveid] << " for delta-E of " << delta_E << endl;
}

// do the stored move
// Caller must make sure to recompute energy!!!!
void Bead::do_move() {
    switch (movetype) {
        case CORNER:
            moveto(linkfrom, prev->linkfrom, prev->neighbor(linkfrom));
            break;
        case CRANKSHAFT1: // recall movedir is w/ respect to pivot, which is prev
            moveto(movedir, linkfrom, prev->neighbor(movedir));
            next->moveto(linkfrom, opposite(movedir),
                         next->next->neighbor(movedir));
            break;
        case CRANKSHAFT2: // movedir is w/respect to pivot, which is next
            prev->moveto(movedir, prev->linkfrom, prev->prev->neighbor(movedir));
            moveto(prev->linkfrom, opposite(movedir), next->neighbor(movedir));
            break;
        default:
            cerr << "do_move on intermediate node " << my_index << ": bad move type " << movetype << endl;
            exit(3);
    }
}

/////////////////////////////////////////////////////////////////////
// CLASSES FirstBead and LastBead derived from Bead
FirstBead::FirstBead(bool newcontents)
    : Bead(newcontents)
{
    movetype = END;
}

// Count interactions with spatial neighbors, but not with adjacent link
int FirstBead::current_E() const {
    int energy = location->interaction_E(contents);
    // take away interactions with next link
    if (contents == next->contents) energy++;
    else energy--;

    return energy;
}

// move FirstBead to new location. Used when making local moves.
void FirstBead::moveto(Direction out, LatticeNode* where) {
    location->unoccupy(contents);
    location = where;
    location->occupy(contents);
    linkfrom = out;
}

// Choose direction for end move. Return success if no collision.
bool FirstBead::generate_move() {
    // Choose any neighbor of pivot except current direction
    Direction exclude = opposite(linkfrom);
    movedir = (Direction)(drand48()*(N_NEIGHBORS - 1)); // 0,1,2,3,4
    if (movedir == exclude) movedir = (Direction)(N_NEIGHBORS - 1); // 5

    // Check for collision
    if (next->neighbor(movedir)->is_occupied()) {
        movedir = INVALID;
        return false;
    }

    // Success!! We'll need delta E
    // both energies contain contribution from adjacent bead,
    // which is cancelled out by subtracting
    delta_E = next->neighbor(movedir)->interaction_E(contents) -
              location->interaction_E(contents);
    return true;
}

// Do the stored move. Caller must recompute energy.
// movedir is direction with respect to pivot
void FirstBead::do_move() {
    moveto(opposite(movedir), next->neighbor(movedir));
}

// DERIVED CLASS LastBead implementation
LastBead::LastBead(int i, bool newcontents, Bead* newprev)
// Count interactions with spatial neighbors, but not with adjacent links
// along the chain
int LastBead::current_E() const {
    int energy = location->interaction_E(contents);
    // take away interactions with prev link
    if (contents == prev->contents) energy++;
    else energy--;

    return energy;
}

// move LastBead to new location.  Used when making local moves.
void LastBead::moveto(Direction in, LatticeNode* where) {
    location->unoccupy(contents);
    location = where;
    location->occupy(contents);
    prev->linkfrom = in;
}

// Choose direction for end move.  Return success if no collision.
bool LastBead::generate_move() {

    // Choose any neighbor of pivot except current direction
    Direction exclude = prev->linkfrom;
    movedir = (Direction)(drand48()*(N_NEIGHBORS - 1));  // 0,1,2,3,4
    if (movedir == exclude) movedir = (Direction)(N_NEIGHBORS - 1);  //

    // Check for collision
    if (prev->neighbor(movedir)->is_occupied()) {
        movedir = INVALID;
        return false;
    }

    // Success!! We'll need delta E
    // both energies contain contribution from adjacent bead,
    // which is cancelled out by subtracting
    delta_E = prev->neighbor(movedir)->interaction_E(contents) -
              location->interaction_E(contents);
    return true;
}

// Do the stored move.  Caller must recompute energy.
// dir is direction with respect to pivot, which is prev
void LastBead::do_move() {
    moveto(movedir, prev->neighbor(movedir));
}
CLASS Chain implementation

// Create a chain and a lattice where it lives.
//   Lattice is expensive to construct. Keep chain for life of
//   program!
//   Chain is created with no location; the creator will need to choose
//   a configuration with either place_straight or Rosenbluth_grow.

Chain::Chain(const char* target)
  : seqlength(strlen(target)), lattice(2*seqlength+2),
    chain_energy(0)
{
  strcpy(seqstring, target);
  seq_from_string(target);

  // construct list of beads
  beads[0] = new FirstBead(sequence[0]);
  for (int i=1; i<seqlength-1; i++) {
    beads[i] = new Bead(i, sequence[i], beads[i-1]);
  }
  beads[seqlength-1] =
    new LastBead(seqlength-1, sequence[seqlength-1], beads[seqlength-2]);
}

// delete array created with new (not new[])
Chain::~Chain() {
  for (int i=0; i<seqlength; i++) {
    delete beads[i];
  }
}

// For initializing from C-string
void Chain::seq_from_string(const char* source) {
  for (int i=0; source[i] != '\0'; i++) {
    switch(source[i]) {
    case 'P':
      sequence[i] = true;
      break;
    case 'H':
      sequence[i] = false;
      break;
    default:
      cerr << "Source " << source << " contains letter other than H,P: 
        " << sequence[i] << endl;
      exit(1);
    }
}
// Place fully extended chain in lattice (zero energy)
void Chain::place_straight() {
    LatticeNode* startnode = lattice.quarternode();
    beads[0]->occupy(startnode);
    for (int pos=1; pos < seqlength; pos++) {
        extend(pos, XPLUS);
    }
    calc_energy();
}

// Write out a configuration, e.g. for saving to file
// (sequence isn't written; it will have to be known!)
// Format: startx starty startz
//          dir0  dir1  dir2 ...
void Chain::write_config(ostream& out) {
    for (int i=0; i < seqlength-1; i++) {
        out << beads[i]->linkfrom << ' ';
    }
    out << endl << endl;
}

// Read in a configuration, e.g. from file. Sequence must be known!
void Chain::read_config(istream& in) {
    clear();
    beads[0]->occupy(lattice.centernode());
    for (int pos = 1; pos < seqlength; pos++) {
        int dir;
        in >> dir;
        extend(pos, (Direction) dir);  // stores bead #pos and link to it
    }
    calc_energy();
}

// copy configuration from a saved array of directions
void Chain::copy_config_from(Direction store[]) {
    clear();
    beads[0]->occupy(lattice.centernode());
    for (int pos = 1; pos < seqlength; pos++) {
        extend(pos, store[pos-1]);
    }
    calc_energy();
}
// copy configuration to an array of directions
void Chain::copy_config_to(Direction store[]) {
    for (int i = 0; i < seqlength-1; i++) {
        store[i] = beads[i]->linkfrom;
    }
}

void Chain::calc_energy() {
    chain_energy = 0;
    for (int i=0; i<seqlength; i++) {
        chain_energy += beads[i]->current_E();
    }
    chain_energy /= 2;  // each interaction was double-counted
}

// Reset to empty lattice and undefined chain (but keep target sequence)
void Chain::clear() {
    lattice.clear();
    for (int i=0; i<seqlength; i++) {
        beads[i]->clear();
    }
    chain_energy = 0;
}

void Chain::print(ostream& out) const {
    if (is_empty()) {
        out << "Empty chain\n";
    } else {
        out << "Chain of length " << seqlength << ", energy " << chain_energy << ": " << endl;
        for (int i=0; i < seqlength-1; i++) { // chain of N beads has N-1 links
            out << seqstring[i] << "--" << direction_name[beads[i]->linkfrom] << "--";
            if (i%8 == 7) out << endl;
        }
        out << seqstring[seqlength-1] << endl;
    }
}

// Print an alternate configuration, rotated to a standard orientation
// in format for Python script to turn into POV file
void Chain::print_config(const Direction config[], ostream& out) const {
    int x=0, y=0, z=0;
    Direction first = config[0];

    for (int i=0; i < seqlength-1; i++) { // chain of N beads has N-1 links
        out << "HP_bead(\" << x << ",\" << y << ",\" << z << ",\n"
<< (sequence[i] ? "'p'" : "'h'") << "," << endl;

    // update position for next bead
switch (ROTATE[first][config[i]]) {
    case XPLUS:
        x++;
        break;
    case XMINUS:
        x--;
        break;
    case YPLUS:
        y++;
        break;
    case YMINUS:
        y--;
        break;
    case ZPLUS:
        z++;
        break;
    case ZMINUS:
        z--;
        break;
    }
    // write final bead
out << "HP_bead(" << x << "," << y << "," << z << ","
    << (sequence[seqlength-1] ? "'p'" : "'h'") << ")" << endl;
}

bool Chain::is_empty() const {
    return (beads[0]->location == 0);
}

int Chain::energy() const {
    return chain_energy;
}

const char* Chain::getstring() const { return seqstring; }
int Chain::getlength() const { return seqlength; }

    // Extend chain forward to bead #pos
    // When pos=3, this is initial config; linkfrom #2 and bead #3 must update
    // Last possible extension is to pos=seqlength-1
    // bead#  0     1     2     3=pos
    //     o-------o----o     x
    // link#     0     1     2
    //        ^
    // | beads[pos-1] has invalid linkfrom
void Chain::extend(int pos, Direction dir) {
    beads[pos-1]->linkfrom = dir;
    beads[pos]->occupy(beads[pos-1]->neighbor(dir));
bool Chain::generate_move() {
    // Pick which bead to move
    movelink = (int)(drand48() * seqlength);

    return beads[movelink]->generate_move();
}

int Chain::deltaE() const { return beads[movelink]->deltaE(); }

void Chain::print_move(ostream& out) const {
    beads[movelink]->print_move(out);
}

void Chain::do_move() {
    beads[movelink]->do_move();
    chain_energy += beads[movelink]->deltaE();
}

// Extend chain by Rosenbluth procedure (ASSUME >1 NODES ALREADY)
// Returns weighting factor (lowercase w in Frenkel) for this extension
// Note: w=0 if chain is backed into corner and can't extend!
// Extends from bead #prev (last_added) to bead #index (to_add)
double Chain::Rosenbluth_extend(int index, bool forward) {
    LatticeNode* candidates[N_NEIGHBORS];
    Direction cand_dirs[N_NEIGHBORS];
    double boltzfacs[N_NEIGHBORS];
    double Z = 0.0;  // partition function for this distribution

    int prev = (forward ? index - 1 : index + 1);
    LatticeNode* endpos = beads[prev]->location;
    bool to_add = sequence[index];
    bool last_added = sequence[prev];

    // make a list of candidate positions
    int ncand = 0;
    for (int i = 0; i < N_NEIGHBORS; i++) {
        LatticeNode* candidate = endpos->neighbor(i);
        if (candidate->is_occupied()) continue; // can't put where
        already full
        candidates[ncand] = candidate;
        cand_dirs[ncand] = (Direction)i;

        // energy that new bead would have, if placed here;
        // not counting interaction energy with adjacent link of chain
        int he = candidate->interaction_E(to_add) + ((to_add == last_added) ? 1 : -1);

        double bf = pow(expmbeta, he);
        Z += bf;
        boltzfacs[ncand] = bf;
    }

    return Z;
}
ncand++;  
}

// If no candidates (all spaces occupied), we're backed into a corner!  
// This configuration is dead!  
if (ncand == 0) return 0.0;

// Pick a winner and extend chain  
int winner = pick(boltzfacs, ncand, Z);  
beads[index]->occupy(candidates[winner]);  
if (forward) {
    beads[index-1]->linkfrom = cand_dirs[winner];
} else {
    beads[index]->linkfrom = opposite(cand_dirs[winner]);
}

return Z;
}

// grow chain by Rosenbluth procedure (randomly forward or backward direction)  
// Might get itself stuck while growing; if so, returns 0, caller should check  
double Chain::Rosenbluth_grow(double temperature) {
    clear();  
    double Rosenbluth_W = 1.0;  
    expmbeta = exp(-1.0/temperature);

    bool forward = (drand48() < 0.5);

    // Put end of chain at center of lattice  
    if (forward) {
        beads[0]->occupy(lattice.centernode());  
        for (int i=1; i < seqlength; i++) {
            Rosenbluth_W *= Rosenbluth_extend(i, forward);
            if (Rosenbluth_W == 0.0) break;  // chain stuck at dead end
        }
    } else {
        beads[seqlength-1]->occupy(lattice.centernode());  
        for (int i=seqlength-2; i >= 0; i--) {
            Rosenbluth_W *= Rosenbluth_extend(i, forward);
            if (Rosenbluth_W == 0.0) break;  // chain stuck at dead end
        }
    }

    // record result  
    calc_energy();  

    return Rosenbluth_W;
}
/* Given a list of weights, pick one of them with probability proportional to weight */

int pick(double weights[], int n, double totweight) {
    double r = drand48()*totweight;
    while (--n > 0) {  // first index is n-1; last is 1 (don't need 0)
        totweight -= weights[n];
        if (r >= totweight) return n;
    }
    return 0;  // last test should always succeed, so don't do it
}
// montecarlo.h  (Local move version)
// Depends on chain.h

#include <iostream>
#include <fstream>
#include "chain.h"

/*******************************************************************************/
// CLASS MonteCarloSim
// Given an initial chain configuration and temperature, the Monte Carlo
// Simulator can run for a given number of steps, or "settle" without
// taking data. It collects an energy histogram.

const int MAXSTORE = 2*MAXLENGTH + 1;

// Each bead has at most 4 interactions with non-chain-linked
// neighbors, so bead's energy ranges from -4 to +4; and crankshaft can move 2
// beads
const int MAXDELTA_E = 2*2*(N_NEIGHBORS - 2);
const int NDELTA_E = 2*MAXDELTA_E + 1;

class MonteCarloSim {
public:
    MonteCarloSim(Chain* c, double temperature);
    void storeboltz();
    void clearhisto();
    void reset_temperature(double newT);

    inline double embeta(int i) const;  // retrieve stored boltzmann
    factors
    void move();
    void settle(int nsteps);
    virtual void run(int nsteps);

    virtual void store(int energy);
    void calc_energy(int steps);
    void write_histogram() const;
    void print_best_config(ostream& out = cout) const;

    int e_min() const;  // for reporting results to caller
    double e_av() const;
    double acceptance() const;  // acceptance during run()
    double heatcap() const;

protected:
    Chain* chain;
    double temperature;

    double expmbeta[NDELTA_E];  // = e^{-beta dE} for each dE
    Direction bestconfig[MAXLENGTH];
int naccepted; // number accepted in current run
int istep;    // number of steps completed in all data runs so far
int curr_energy;
int min_energy;
int Ecounts[MAXSTORE]; // histogram

double curr_avenergy;   // at last computation; computed in run()
double curr_varenergy;
double curr_acceptance;  // in last run() only

private:
    // birth control for unwanted functions
    MonteCarloSim();
    MonteCarloSim(const MonteCarloSim& rhs);
    MonteCarloSim& operator=(const MonteCarloSim& rhs);
};

//////////////////////////////////////////////////////
// CLASS NativeSeeker inherits from MonteCarloSim
//
// This simulation looks for the native (i.e. minimum energy) state
// and stops when it finds it.
// Optionally it may report energy at given intervals, for looking at
// trajectories.

class NativeSeeker : public MonteCarloSim {
public:
    NativeSeeker(Chain* c, double temp, int goal);
    NativeSeeker(Chain* c, double temp, int goal,
        ofstream& output, int interval);

    // override base class
    virtual void run(int nsteps);
    void run(int nsteps, int block, ofstream& log);
    virtual void store(int energy);

    // new functions
    int get_time() const;
    bool was_found() const;

private:
    const int Enative;
    bool found;
    int found_time;
};
// montecarlo.cpp

#include "montecarlo.h"

#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>  // for drand48()
#include <cmath>

// CLASS MonteCarloSim implementation

MonteCarloSim::MonteCarloSim(Chain* c, double temp)
    : chain(c), temperature(temp),
      naccepted(0), istep(0),
      curr_energy(chain->energy()), min_energy(curr_energy)
{
    storeboltz();
    clearhisto();
    // save the initial configuration
    chain->copy_config_to(bestconfig);
}

void MonteCarloSim::storeboltz() {
    // calculate all the Boltzmann factors at once
    for (int i = -MAXDELTA_E; i <= MAXDELTA_E; i++) {
        expmbeta[i + MAXDELTA_E] = exp(-i/temperature);
    }
}

void MonteCarloSim::clearhisto() {
    // clear histogram
    for (int i = 0; i < MAXSTORE; i++) {
        Ecounts[i] = 0;
    }
}

// retrieve saved value of e^{-beta dE}
double MonteCarloSim::embeta(int i) const {
    return expmbeta[i + MAXDELTA_E];
}

// reset temperature and prepare for new run; used for simulated annealing
void MonteCarloSim::reset_temperature(double newT) {
    temperature = newT;
    storeboltz();
    clearhisto();
}
void MonteCarloSim::move() {
    int delta_E;

    if (chain->generate_move()) { // Reject if can't generate legal move
        #ifdef TEST2
            cout << "Step " << istep << " : ";
            chain->print_move();
        #endif
        delta_E = chain->deltaE();

        // Metropolis criterion
        if ( delta_E <= 0 ||                   // accept immediately
            drand48() < embeta(delta_E) ) {  // accept with some probability
            chain->do_move();
        #ifdef TEST2
            cout << "ACCEPTED!!\n";
        #endif
        #ifdef TEST1
            chain->print();
        #endif
            curr_energy += delta_E;
            naccepted++;
        }
    }
}

void MonteCarloSim::settle(int nsteps) {
    cout << "Monte Carlo settling at kT = " << temperature << " for " << nsteps << " steps" << endl;
    for (int istep = 0; istep < nsteps; istep++) {
        move();
    }
}

void MonteCarloSim::run(int nsteps) {
    const int MILLION = 1000000;
    cout << "Monte Carlo run at kT = " << temperature << " for " << nsteps << " steps" << endl;
    cout << setiosflags(ios::left) << setw(8) << "Million" << "Min" << endl
         << setw(8) << "steps" << setw(8) << "energy"
         << setw(10) << "<E>
         << setw(10) << "<E^2>"
         << setw(10) << "accepted" << endl;
    cout << "---------------------------------------------" << endl;
naccepted = 0;  // want acceptance rate during this run

// chain might have been reset since last time, so
curr_energy = chain->energy();

// loop in blocks of 1 million
// recall istep was initialized to 0 when Simulator was created
int remaining = nsteps;
int init = istep;  // to keep track of how far we've gone in this run
while (remaining > 0) {
    int block = MILLION;
    if (block > remaining) block = remaining;
    for (int i=0; i < block; i++) {
        move();
        store(curr_energy);
        istep++;
    }
    remaining -= block;

    // output statistics after every million steps
    calc_energy(istep);
    curr_acceptance = (double)naccepted/(istep-init);
    cout << setw(8) << istep/MILLION
         << setw(8) << min_energy
         << setiosflags(ios::fixed) << setprecision(3)
         << setw(10) << curr_avenergy
         << setw(10) << curr_varenergy
         << curr_acceptance*100.0 << '%' << endl;
}

// and we're done
calc_energy(istep);
double Cv = (curr_varenergy - curr_avenergy*curr_avenergy)/
           (temperature*temperature);
cout << "\nHeat capacity: " << Cv << endl;
*/
cout << "\nBest configuration seen:\n";
chain->print_config(bestconfig);
*/
write_histogram();
}

const int OFFSET = MAXLENGTH;   // lowest energy we'll see is - MAXLENGTH
void MonteCarloSim::store(int e) {
    Ecounts[e + OFFSET]++;
    if (e < min_energy) {
        // cout << "New best: " << e << " at step " << istep << endl;
        min_energy = e;
        chain->copy_config_to(bestconfig);
    }
}
void MonteCarloSim::calc_energy(int steps) {
    double totenergy = 0.0, totenergysq = 0.0;
    for (int e=0; e<MAXSTORE; e++ ) {
        int energy = e - OFFSET;
        totenergy += (double)energy * Ecounts[e];
        totenergysq += (double)energy * energy * Ecounts[e];
    }
    curr_avenergy = totenergy/steps;
    curr_varenergy = totenergysq/steps;
}

void MonteCarloSim::write_histogram() const {
    int max_energy = min_energy;
    int energy;
    for (energy = min_energy; energy < MAXLENGTH; energy++) {
        if (Ecounts[energy + OFFSET] > 0) max_energy = energy;
    }
    cout << "Energy Histogram
-----------------
" << "E	Counts
------------------
" << endl;
    for (energy = min_energy; energy <= max_energy; energy++) {
        cout << setiosflags(ios::right) << setw(3) << energy << '	'
             << Ecounts[energy + OFFSET] << endl;
    }
}

void MonteCarloSim::print_best_config(ostream& out) const {
    out << "Best configuration: energy " << min_energy << endl;
    chain->print_config(bestconfig, out);
}

int MonteCarloSim::e_min() const {  return min_energy; }
double MonteCarloSim::e_av() const {  return curr_avenergy; }
double MonteCarloSim::acceptance() const {  return curr_acceptance; }


// CLASS NativeSeeker implementation
NativeSeeker::NativeSeeker(Chain* c, double temp, int goal)
    : MonteCarloSim(c, temp),
      Enative(goal), found(false), found_time(0)
}
void NativeSeeker::store(int e) {
    Ecounts[e + OFFSET]++;
    if (e < min_energy) {
        // cout << "New best: " << e << " at step " << istep << endl;
        min_energy = e;
        chain->copy_config_to(bestconfig);
        if (min_energy == Enative) found = true;
    }
}

void NativeSeeker::run(int nsteps) {
    naccepted = 0;
    // loop until found
    found = false;
    int istep;
    for (istep = 0; istep < nsteps; istep++) {
        move();
        store(curr_energy);
        if (found) break;
    }
    // and we're done
    calc_energy(istep);
    curr_acceptance = (double)naccepted/istep;
    // caller will repeat run() many times and write header
    cout << setiosflags(ios::fixed) << setw(5) << setprecision(3) << temperature
        << setw(10) << istep
        << setw(8) << min_energy
        << setw(8) << curr_avenergy
        << setw(11) << setprecision(2) << 100.0*curr_acceptance << '%';
    if (found) {
        found_time = istep;
        cout << " **Native state found, step " << istep;
    }
    cout << endl;
}

void NativeSeeker::run(int nsteps, int block, ofstream& logfile) {
    naccepted = 0;
    // loop until found in blocks of given size
    found = false;
    int istep = 0;
    int remaining = nsteps;
    while (remaining > 0) {
        move();
        store(curr_energy);
        if (found) break;
        calc_energy(istep);
        curr_acceptance = (double)naccepted/istep;
        // caller will repeat run() many times and write header
        cout << setiosflags(ios::fixed) << setw(5) << setprecision(3) << temperature
            << setw(10) << istep
            << setw(8) << min_energy
            << setw(8) << curr_avenergy
            << setw(11) << setprecision(2) << 100.0*curr_acceptance << '%';
        if (found) {
            found_time = istep;
            cout << " **Native state found, step " << istep;
        }
        cout << endl;
        remaining -= block;
    }
}
if (block > remaining) block = remaining;
for (int i=0; i < block; i++) {
    move();
    store(curr_energy);
    istep++;
    if (found) break;
}
remaining -= block;

// output position after every block, or after finding native state
logfile << curr_energy << endl;
if (found) break;

// and we're done
calc_energy(istep);
curr_acceptance = (double)naccepted/istep;

// caller will repeat run() many times and write header
cout << setiosflags(ios::fixed) << setw(5) <<
    setprecision(3) << temperature <<
    setw(10) << istep <<
    setw(8) << min_energy <<
    setw(8) << curr_avenergy <<
    setw(11) << setprecision(2) << 100.0*curr_acceptance << '%';
if (found) {
    found_time = istep;
    cout << " **Native state found, step " << istep;
}
cout << endl;

int NativeSeeker::get_time() const { return found_time; }
bool NativeSeeker::was_found() const { return found; }
// simanneal.cpp

// Perform a simulated annealing run starting at kT=1.0 until goal is reached
// usage: simanneal string alpha goal [Nsteps] [maxstages] [block, logfile]
// Nsteps = number of steps per stage

#include <iostream>
#include <iomanip>
#include <sys/types.h>  // for time
#include <time.h>
#include <cmath>
#include <cstdlib>
#include "montecarlo.h"

int main(int argc, char* argv[]) {

    // seed random number generator with current time
    srand48((long)time(0));

    int DEFAULT_N_STEPS = 10000000;
    int DEFAULT_MAX_STAGES = 100;
    int nsteps, maxstages, block;
    char* logfilename;
    bool logging = false;

    char* seqstring = argv[1];
    double alpha = atof(argv[2]);
    int goal = atol(argv[3]);

    if (argc < 4 || argc > 8) {
        cerr << "Bad command; should be
        " << "simanneal string alpha goal Nsteps maxstages block
        " << "(last 4 arguments are optional)\
        " << "logfile\n        " << "(last 4 arguments are optional)\n"
        exit(1);
    }
    nsteps = DEFAULT_N_STEPS;
    maxstages = DEFAULT_MAX_STAGES;

    if (argc > 4) {
        nsteps = atol(argv[4]);
    }
    if (argc > 5) {
        maxstages = atol(argv[5]);
    }
    if (argc > 6) {
        block = atol(argv[6]);
        logfilename = argv[7];
        logging = true;
    }
}
const double TSTART = 1.0;
double temperature = TSTART;
Chain *chain = new Chain(seqstring);
chain->Rosenbluth_grow(temperature);
cout << "Initialized chain of length " << chain->getlength()
   << " by Rosenbluth growth at temperature " << temperature << endl;

NativeSeeker seeker(chain, TSTART, goal);
ofstream logfile;
if (logging) {
    logfile.open(logfilename);
    if (logfile.fail()) {
        cerr << "Couldn't open " << logfilename << " for output,
             quit\n";
        exit(1);
    }
    logfile << "Samples from simulated annealing at intervals of "
            << block << " steps\n";
}

// print header for output from each Monte Carlo run
cout << setw(5) << "kT" << setw(10) << "Nsteps"
    << setw(8) << "E_min" << setw(8) << "<E>" << setw(12) << "acceptance"
    << endl
    << "--------------------------------------------------" << endl;
bool found = false;
int stage;
for (stage = 1; stage <= maxstages; stage++) {
    if (logging) {
        seeker.run(nsteps, block, logfile);
    } else {
        seeker.run(nsteps);
    }
    if (seeker.was_found()) {
        found = true;
        break;
    } else {
        temperature *= alpha;
        seeker.reset_temperature(temperature);
    }
}
if (logging) logfile.close();
cout << endl;
if (found) {
    cout << "Found native state during stage #" << stage << ", T=
         " << temperature << endl;
    seeker.print_best_config();
} else {
    cout << "Completed " << maxstages << " temperature stages"
         << " without finding native state\n";
}
    return 0;
}
// heatcap.cpp
// usage: heatcap sequence temperature settling-steps run-steps
//   infile contains an initial sequence

#include <iostream>
#include <fstream>
#include <sys/types.h>  // for time
#include <time.h>
#include <cmath>
#include <cstdlib>
#include "montecarlo.h"

int main(int argc, char* argv[]) {
    // seed random number generator with current time
    srand48((long)time(0));

    if (argc < 5 || argc > 6) {
        cerr << "heatcap: bad command line; should be
        "heatcap sequence temperature settle-steps run-steps
        [infile]"
        << endl;
        exit(1);
    }
    char* seqstring = argv[1];
    double temperature = atof(argv[2]);
    int nsettle = atol(argv[3]);
    int nrun = atol(argv[4]);

    bool readin = false;
    ifstream infile;
    if (argc == 6) {
        readin = true;
        infile.open(argv[5]);
        if (infile.fail()) {
            cerr << "Unable to open " << argv[5] << " for input,
            quitting\n"
            << endl;
            exit(1);
        }
    }
    Chain *chain = new Chain(seqstring);
    if (readin) {
        chain->read_config(infile);
    }
    else {
        chain->Rosenbluth_grow(temperature);
    }

    MonteCarloSim sim(chain, temperature);
sim.settle(nsettle);
sim.run(nrun);

/*
   cout << "\nOverall average heat capacity: " << sim.heatcap() <<
   endl;
   sim.print_best_config();
*/

    return 0;
}
// firstpassage.cpp

// Look for first passage at given temperature
// usage: firstpassage string temperature nsettle maxsteps

#include <iostream>
#include <iomanip>
#include <sys/types.h>  // for time
#include <time.h>
#include <cmath>
#include <cstdlib>
#include "montecarlo.h"

int main(int argc, char* argv[]) {
    // seed random number generator with current time
    srand48((long)time(0));

    if (argc !=5) {
        cerr << "Bad command; should be"
        << "firstpassage string temperature nsettle maxsteps" << endl;
        exit(1);
    }
    char* seqstring = argv[1];
    double temperature = atof(argv[2]);
    int nsettle = atol(argv[3]);
    int maxsteps = atol(argv[4]);

    Chain *chain = new Chain(seqstring);
    chain->Rosenbluth_grow(temperature);
    cout << "Initialized chain of length " << chain->getlength()
    << " by Rosenbluth growth at temperature " << temperature << endl;
    NativeSeeker seeker(chain, temperature, -36);
    seeker.settle(nsettle);

    // print header for output from each Monte Carlo run
    cout << setw(5) << "kT" << setw(10) << "Nsteps"
    << setw(8) << "E_min" << setw(8) << "<E>" << setw(12) << "acceptance"
    << endl
    << "--------------------------------------------------" << endl;
    seeker.run(maxsteps);
    cout << endl;
    if (seeker.was_found()) {
        cout << "**** Reached native state at step " << seeker.get_time()
        << endl;
    } else {
        // code continues here
cout << "Failed to find native state after " << maxsteps
    << " steps" << endl;
}

return 0;
}
Minimum energy configuration of sequence 1:

(0,0,0,'h'), (1,0,0,'h'), (2,0,0,'p'), (3,0,0,'h'), (3,0,1,'p'), (2,0,1,'p'), (1,0,1,'h'), (0,0,1,'h'), (0,0,2,'h'), (1,0,2,'h'), (2,0,2,'p'), (3,0,2,'p'), (3,1,2,'p'), (2,1,2,'p'), (1,1,2,'h'), (1,1,1,'h'), (1,2,1,'p'), (0,2,1,'h'), (0,2,0,'p'), (0,1,0,'h'), (0,1,1,'h'), (0,1,2,'h'), (0,2,2,'h'), (1,2,2,'h'), (2,2,2,'p'), (2,2,1,'p'), (2,1,1,'p'), (3,1,1,'p'), (3,1,0,'h'), (2,1,0,'p'), (1,1,0,'h'), (1,2,0,'p'), (2,2,0,'p'), (3,2,0,'h'), (3,2,1,'p'), (3,2,2,'p')

Sequence 2 configuration A:
(0,0,0,'h'), (1,0,0,'h'), (2,0,0,'h'), (2,0,-1,'p'), (1,0,-1,'p'), (0,0,-1,'h'), (0,-1,-1,'h'), (1,-1,-1,'p'), (1,-1,0,'h'), (0,-2,0,'h'), (1,-2,0,'h'), (1,-2,-1,'p'), (0,-2,-1,'h'), (0,-2,-2,'h'), (1,-2,-2,'p'), (1,-1,-2,'p'), (0,-1,-2,'h'), (0,-1,-3,'p'), (0,0,-3,'h'), (0,0,-2,'h'), (1,0,-2,'p'), (1,0,-3,'h'), (1,-3,'p'), (2,-1,-3,'p'), (2,0,-3,'h'), (2,0,-2,'p'), (2,-1,-2,'p'), (2,-1,-1,'p'), (2,-1,0,'h'), (2,-2,0,'h'), (2,-2,-1,'p'), (2,-2,-2,'p'), (2,-2,-3,'p'), (1,-2,-3,'p'), (0,-2,-3,'p')

Sequence 2 configuration B:
(0,0,0,'h'), (1,0,0,'h'), (1,0,-1,'h'), (1,0,-2,'p'), (1,1,-2,'p'), (1,1,-1,'h'), (1,1,0,'h'), (1,1,1,'p'), (0,1,1,'h'), (0,1,0,'h'), (0,2,0,'h'), (0,2,1,'h'), (1,2,1,'p'), (1,2,0,'h'), (1,2,1,'h'), (1,2,-2,'p'), (0,2,-2,'p'), (0,2,-1,'h'), (-1,2,-1,'p'), (-1,2,0,'h'), (-1,2,1,'p'), (-1,1,0,'h'), (-1,1,-1,'p'), (0,1,-1,'h'), (0,1,-2,'p'), (0,0,-2,'p'), (0,0,-1,'h'), (-1,0,-1,'p'), (-2,0,-1,'p'), (-2,1,-1,'p'), (-2,1,0,'h'), (-2,2,0,'h'), (-2,2,-1,'p'), (-2,2,-2,'p'), (-1,2,-2,'p'), (-1,1,-2,'p'), (-1,0,-2,'p')

Sequence 2 configuration C:
(0,0,0,'h'), (1,0,0,'h'), (1,1,0,'h'), (1,2,0,'p'), (1,2,1,'p'), (1,1,1,'h'), (2,1,1,'h'), (2,0,1,'p'), (1,0,1,'h'), (0,0,1,'h'), (0,0,2,'h'), (1,0,2,'h'), (2,0,2,'p'), (2,1,2,'h'), (1,1,2,'h'), (1,2,2,'p'), (0,2,2,'p'), (0,1,2,'h'), (-1,1,2,'p'), (-1,0,2,'h'), (-1,0,1,'h'), (-1,1,1,'p'), (0,1,1,'h'), (0,2,1,'p'), (0,2,0,'p'), (0,1,0,'h'), (-1,1,0,'p'), (-2,1,0,'p'), (-2,1,1,'p'), (-2,0,1,'h'), (-2,0,2,'h'), (-2,1,2,'p'), (-2,2,2,'p'), (-2,2,1,'p'), (-1,2,1,'p'), (-1,2,2,'p')