Don't Tump Over: A Heuristic Cargo Hold Model

New Mexico
Supercomputing Challenge
Final Report
April 4th, 2007

Team 032
Manzano High School
Eldorado High School

Team Members
Sam Boling, Manzano High School
William Laub, Eldorado High School

Teachers
Stephen Schum, Manzano High School
Jennifer Keller, Eldorado High School

Project Mentor
Tom Laub

Table of Contents

Executive Summary

We set out to computationally model a two-dimensional cargo hold and find adequate methods for loading any given set of rectangular packages. Due to the problem's computational complexity (it constitutes an instance of the knapsack problem and is documented as NP-hard[1,7]), it is necessary to approximate loading methods. Our program devises "good" loading sequences (defined as those that are acceptably fast, load as many packages as possible, and balance the center of mass in the center of the hold) comprised of percent chances to use one of several computationally simple loading methods.

The program first populates a list of packages with lengths, widths, and masses. From these characteristics it determines characteristics such as density and "squariness," or the closeness of a package's dimensions to those of a square package. It then generates a self-organizing map, or SOM, a type of neural network used as a search algorithm. A self-organizing map contains a number of output nodes and a few input nodes, each with certain attributes. In our case, node attributes are percent-chances to use one of several computationally simple loading methods. A package may be loaded using one of several different "greedy approximations," which prioritize packages in accordance with their characteristics. A greedy approximation by mass, for example, first loads the heaviest package, then the next heaviest, and so on. Each node, then, stores five percent-chances from 0 to 100: greedy approximations by mass, area, density, and squariness, and a chance to pick an unloaded package at random and load it.

Problem Statement

It was our intention to develop a program capable of generating loading sequences for any set of rectangular packages into a rectangular hold, optimizing for speed, fit, and balance. However, when we want to fit any set of objects into a limited space, we encounter the "Knapsack problem,"[1,7] a classic mathematical problem that is notoriously difficult to solve computationally. The Knapsack problem is documented as NP-Hard, meaning that it is at least as hard as every problem in the set NP. NP is a set containing all problems which can be solved in polynomial time by a non-deterministic Turing machine. Problems solvable in polynomial time are those which are solvable in as many steps as it takes to solve a polynomial of the "problem size," or size of the input; in other words, these are problems that can be solved "fast."[2,5] Turing machines are computational "black boxes" which process input and produce output. In a deterministic Turing machine, one input is always mapped to exactly one output. In a non-deterministic Turing machine, this is not necessarily the case: some other factor affects the output.[6] It becomes necessary, because of the computational complexity of the Knapsack problem, to use a dynamic programming approach.

Solution Method

Dynamic programming refers to the branch of programming that deals with finding reductions of complex problems; that is, dynamic solutions entail solving small divisions of a problem which, when combined in a significant way, yield the solution to the problem as a whole.[8] In our case, we are interested in approximating "good" loading methods in a Monte Carlo fashion: by randomly generating our initial SOMs and using probabilities to generate a mix of loading methods, we achieve a stochastic solution to this complex problem. We use an algorithm called a Self-Organizing Map, or SOM, to approach desirable loading methods.

SOMs first randomly generate the characteristics for an array of nodes laid out in two or more dimensions. They then compare these characteristics to the characteristics of several inputs. The node which is most similar to an input node is called that input node's best matching unit, or BMU. Once the BMU for an input has been found, the SOM updates all nodes by the following formula, making nodes within a certain radius more similar to the BMU. As a result, the SOM becomes more similar to the inputs with each iteration.

For each input characteristic value *X* at iteration *n*,

$$X_{n+1} = X_n + \theta(n)\alpha(n)(d - X_n)$$

Θ is the "neighborhood function," a hyperbolically decreasing value that describes the distance from the BMU within which it is acceptable to update. **α** is a monotonically decreasing function that reduces the magnitude of the update as time progresses. **d** is the Euclidean distance ($\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$) between the node in question and the BMU. In our case, $\Theta(x)$ is given by

$$\Theta(x) = \frac{a}{(x+1)^n} + 3$$

where ***a*** is the width (in nodes) minus three.

α (x) is given by

$$\alpha(x) = \frac{1}{x + 4.25} + \frac{3}{4}$$

We use this type of hyperbolic function because the learning coefficient must decrease monotonically and never be less than zero. If the learning coefficient were to drop below zero, the changes would have the opposite of the intended effect and would begin to result in massive changes as the magnitude of the coefficient increased. The hyperbolic function has an asymptote which can be easily positioned. We started using a simple hyperbolic function with the asymptote at zero and then began modifying the asymptote so that the learning coefficient did not become so low that changes became insignificant, but also did not stay so high that the SOM never stabilized. The tailoring of the learning function was accomplished during the development of our Red, Green, Blue (RGB) SOM. We used the RGB SOM to become familiar with SOMs because we had an idea of what the results should be and therefore could gauge the correctness of our SOM algorithm. The resulting function has an asymptote at 0.5. Figures 1 through 3 show some results of our initial work with the RGB SOM.

Using these formulas, we update a random set of nodes with a random set of inputs containing the same type of data as the nodes. Our node characteristics represent percentage chances to use one of several computationally-simple loading methods. It is a simple matter to find, for example, the heaviest package, to load it first, and then to load the next heaviest package and so on. This is known as a greedy approximation by mass. Each node in our SOM stores five characteristics: the percent chance to greedily approximate by size, the percent chance to greedily approximate by mass, the percent chance to greedily approximate by density, the percent chance to greedily approximate based on the similarity of the package shape to a square, and the percent chance to pick a package at random from the list of unloaded packages and load it. These characteristics comprise what we identify as a "loading method." Each loading method has a probability assigned to it such that the probabilities for all methods add up to 100. We then select loading methods by their probabilities in order to create a unique and computationally simple combination of the less desirable greedy approximations. At each iteration, we test our nodes in this way to determine the "fitness" of each loading method, or an average of speed, fit, and balance. The few, the proud, the five nodes with the greatest fitness are used as the inputs for the next iteration. We repeat this process until the improvement in fitness from step to step is very slight, at which point we halt execution and output the loading method with the greatest fitness.

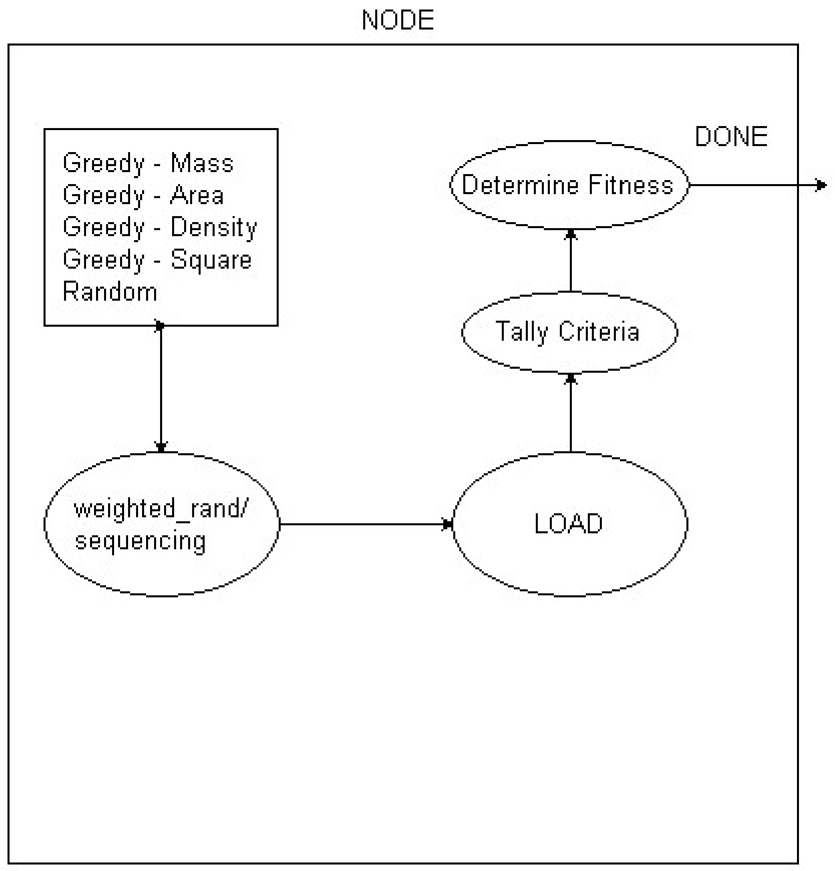
Loading simulation

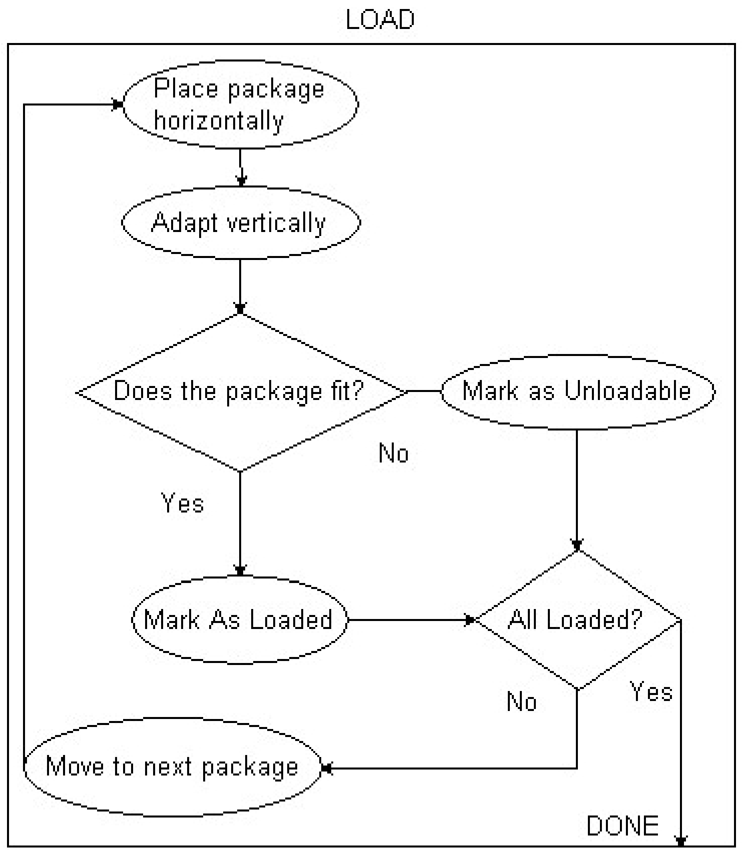
In order to evaluate loading sequences, we must be able to simulate their results. To do this, we simulate the loading of the packages using what we call the “Tetris” method. We place our first package at the coordinates (0,0) and then continue to load packages along the increasing x-axis such that they’re vertical edges are aligned. After selecting the x position, we check along the side of the package closest to y = 0 to see if it is closer to y = 0 than any of the previously loaded packages that could intersect it. If any packages are detected meeting those conditions, the package is moved such that it can no longer intersect. As a result, the hold loads much like a game of Tetris: the packages stack on top of each other and “drop” into the hold from left to right. During the loading process, if any package exceeds the boundaries of the hold, it is marked as unloadable. In our visualization, the y-axis increases toward the bottom of the screen and the x-axis increases toward the right, so the first package appears at the top left corner of the hold and the other packages “fall” upward from left to right. A sample of our loading visualization appears in Figure 4. In Figure 4, a set of 75 packages has been loaded into a hold by size. It was sequenced using probabilities of 0 for all methods except greedy approximation by size. The brown packages were all loaded successfully and the red packages did not fit. The white dot near the center shows the center of mass and the diagonal lines increase in length for packages loaded later in the loading sequence. In this load, the center of mass would have been favorable, but the packages haven’t all been loaded, so the sequence is not sufficient. We used the SDL[10] (Simple DirectMedia Layer) API to produce our loading visualizations.

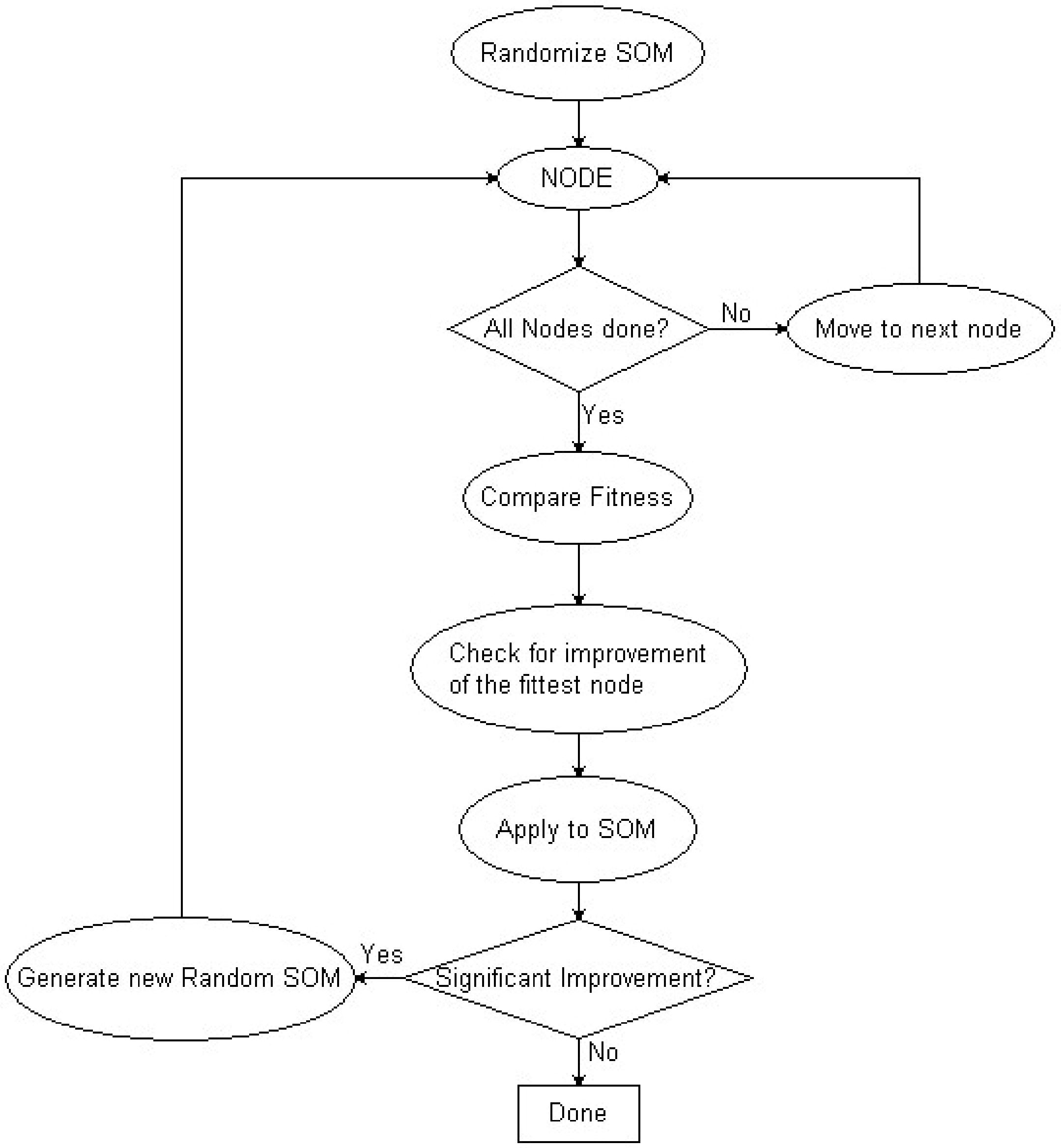
Results

As of April 3, 2007, we have created a working SOM, which we initially created as the RGB (Red, Green, and Blue) SOM used in our examples and adapted to function in our final implementation. We have also created functions to sequence the packages using data from the SOM, apply the “Tetris” loading simulation to those packages, and display the loaded hold with loaded and unloadable packages. Unfortunately, during the marriage of the loading algorithms and the SOM we introduced bugs. We are currently debugging the combination of the two and expect to have results on the 2D implementation of the full program by Expo. A flow chart of our code as it stands now is shown below.

Flowchart




LOAD
Place package horizontally
Adapt vertically
Does the package fit?
No
Mark as Unloadable
Yes
Mark As Loaded
All Loaded?
No
Yes
Move to next package
DONE


Randomize SOM
NODE
All Nodes done?
No
Move to next node
Yes
Compare Fitness
Check for improvement of the fittest node
Apply to SOM
Significant Improvement?
Yes
Generate new Random SOM
No
Done

Figures (and grounds)

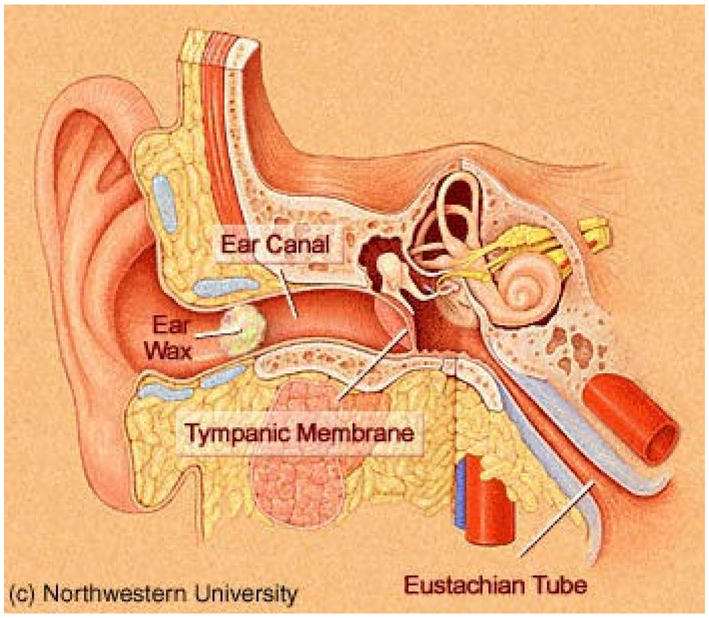
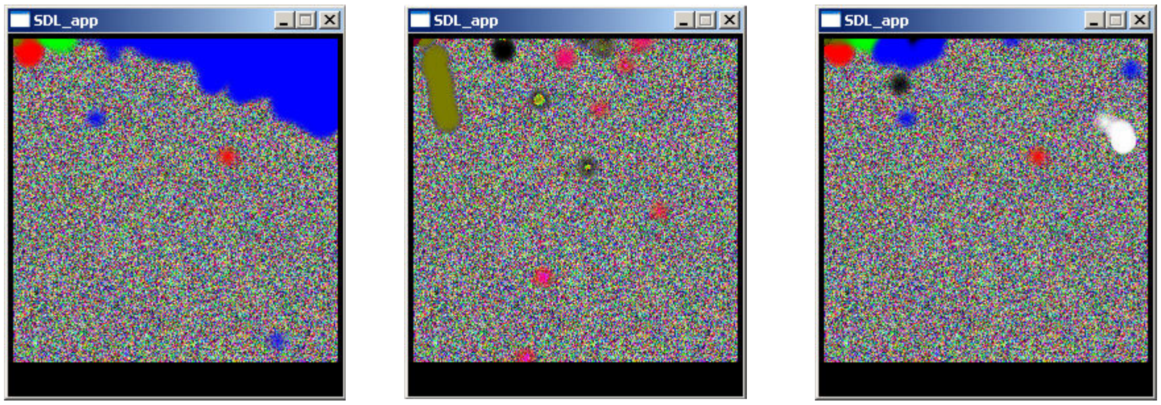


Figure 0. An ear with earwax. The earwax represents the moral and economical blockage caused by the complexity of cargo loading. Our project is represented by the off-screen water pick preparing to clear the blockage and revive the ethical and financial ear of the nation.

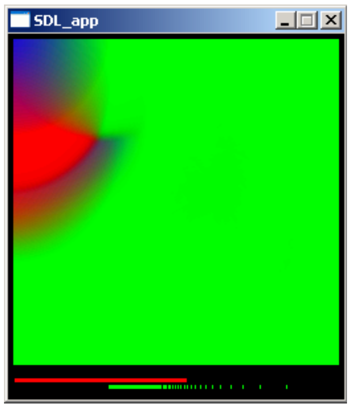


Input: Red, Blue, Green

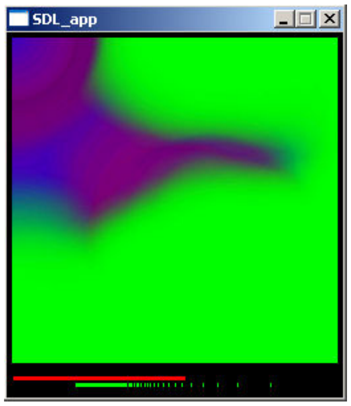
Input: Black, Pink, Dark Green

Input: Red, Blue, Green, White, Black

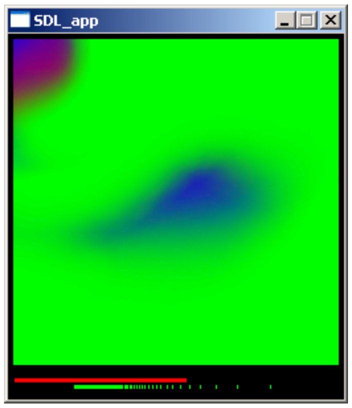
Figure 1. A few examples of our RGB SOM before implementing a changing update radius.



n = .25

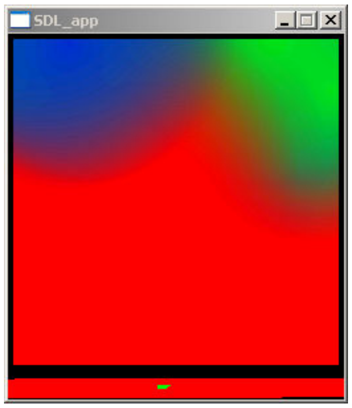


n = .33



n = .34

Figure 2. Our RGB SOM after implementing a decreasing update radius. In these images, **n** is the exponent in the denominator of the function. Each of these starts the same and runs for 130 iterations using red, green, and blue as the inputs. The red bar at the bottom of each view helps to monitor the progress of the SOM, with one pixel added to its length for each iteration. The green markings at the bottom show the update radius and move left over time as the update radius decreases.



n = 0.1

Figure 3. This SOM ran for 3250 iteration and stabilized as it appears here.

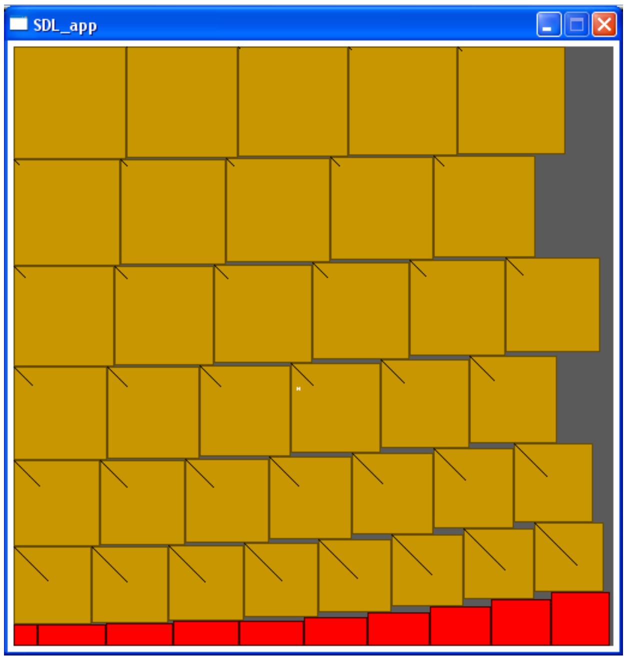


Figure 4. In this image, a set of 75 packages has been loaded into a hold by size. It was sequenced using probabilities of 0 for all methods except greedy approximation by size. The brown packages were all loaded successfully and the red packages did not fit. The white dot near the center shows the center of mass and the diagonal lines increase in length for packages loaded later in the loading sequence. In this load, the center of mass would have been favorable, but the packages haven't all been loaded, so the sequence is not sufficient.

Code Documentation

Classes

```
class package
{
  public:
     package();
```

This is the most important part of the program. It is the constructor for the package class.

```
     Void setmass(float setm);
     void setside1(float sets1);
     void setside2(float sets2);
     void setlocation(float x, float y);
     void setunloadable();
```

This is the most important part of the program. The above are the set functions for private variables.

```
     Void load();
     void unload();
```

This is the most important part of the program. These functions set and unset the `loaded` variable.

```
     Float getMass();
     float getSide1();
     float getSide2();
     float getDensity();
     float getArea();
     float getWeightClass();
     float getSquariness();
     float getX();
     float getY();
     short checkLoaded();
```

This is the most important part of the program. These functions retrieve private variables

```
  private:
     float side1;
     float side2;
     float squariness;
     float mass;
     float density;
     float xpos;
     float ypos;
     short loaded;
};

typedef struct point {
               int xpos;
               □n typos;
        };
```

This is the most important part of the program. It defines a simple data structure used to consolidate variables in some important functions.

```
Class node{
public:
       unsigned int getXPos();
       unsigned int getYPos();
       unsigned int getgMass();
       unsigned int getgArea();
       unsigned int getgSqrn();
       unsigned int getgDens();
       unsigned int getrandom();
       unsigned int getOK();
       float getFitness();
       unsigned int getSteps();
       unsigned int loadedOK();
```

This is the most important part of the program. These are the get functions for the `node` class's private variables

```
       void setLoc(unsigned int x, unsigned int y);
       void setWeights(unsigned int mass, unsigned int area,
unsigned int squariness, unsigned int density, unsigned int
random_p);
       void setOK(unsigned short value);
       void setFitness(float value);
       void incSteps();
       void incLoaded();
```

This is the most important part of the program. These functions modify private variables of the `node` class.

```
       Void load_package(short number, package *pkgs);
```

`load_package` is the most important part of the program. It loads the `number`th package in the set `*pkgs` into the hold. Packages are first loaded from left to right, then checks are made along the top to make sure that packages don't overlap or go outside the hold.

```
Void sequence(short *by_weight, short *by_size,     short
*by_squariness, short *by_density, package *order,
package *pkgs, short weights[], short number, short
weightnum);
```

`sequence` is the most important part of the program. It generates, based on their orders by size, mass, density and squariness, an order for the packages based on the probabilities from the node in question.

```
Float FindCOM(package *pkgs, short number, float*
COMx, float* COMy);
```

`FindCOM` is the most important part of the program. It locates the center of mass of all the loaded packages combined based on their loaded positions.

```
Private:
       unsigned int xpos;
       unsigned □n typos;
       unsigned int gMass,gArea,gSqrn,gDens,random;
       float fitness;
       unsigned int timesteps;
       unsigned int numLoaded;

       unsigned short okToUpdate;
};
```

Functions

```
short weighted_rand(short *weights, short number);
```

`weighted_rand` is the most important part of the program. It takes an array of "weights," or percent probabilities, and the length of that array. It returns a number that corresponds to the percent probability in which the randomly generated number fell. For example, if the function were passed an array containing 40, 20, 10, and 30, the number falling in the 20% range would return 2.

```
Void make_hold(float width, float height);
```

`make_hold` is the most important part of the program. It generates the important characteristics of the hold, such as MAXW, MAXH, and the position of the center.

```
Void make_pkgs(float *width, float *height, float *weight, float
*squariness, float *density, short *weight_order, short
*size_order, short *beide_order, short *squariness_order, short
*density_order, short number);
```

`make_pkgs` is the most important part of the program. It generates the packages' length, width, and mass, and orders them by weight, size, density and squariness.

```
Void draw_rect(SDL_Surface *screen, float width, float height,
float xpos, float ypos,  short r, short g, short b);
```

`draw_rect` is the most important part of the program. It draws a rectangle on a surface given the x and y positions of the top right corner, the width and height, and the red, green, and blue color values.

```
Void show(package *pkgs, SDL_Surface *screen, short number);
```

`show` is the most important part of the program. It draws onto the screen the packages that have been loaded.

```
Void DrawPixel(SDL_Surface *screen, int x, int y, Uint8 R, Uint8
G, Uint8 B);
```

`DrawPixel` is the most important part of the program. It draws a pixel at a specified x and y position and red, green, and blue values.

```
Void Slock(SDL_Surface *screen);
```

`Slock` is the most important part of the program. It locks the screen for drawing.

```
Void Sulock(SDL_Surface *screen);
```

`Sulock` is the most important part of the program. It unlocks the screen.

Code Listing

Below is a complete listing of our code. We have tried to minimize the unintentional wrapping of lines by reducing the font size; however, some line-wrapping still occurs. We hope it is obvious where this is important.

main.cpp:

```
#include <cstdlib>
#include <iostream>
#include <cmath>
#include "functions.h"
#include "package2d.h"
#include "globalvars.h"
#include "node.h"
#include "som.h"
#include <SDL.h>

using namespace std;

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

    short *weight_order;
    short *size_order;
    short *squariness_order;
    short *density_order;

    package *pkgs;
    short number = 75;
    package *order;
    short orderct = 75;

    make_hold(500,500);

//////////////////////
//SDL initialization//
//////////////////////

    if ( SDL_Init(SDL_INIT_AUDIO|SDL_INIT_VIDEO) < 0 )
    {
      printf("Unable to init SDL: %s\n", SDL_GetError());
      exit(1);
    }
    atexit(SDL_Quit);

    SDL_Surface *screen;

screen=SDL_SetVideoMode((short)hold_sedge+10,(short)hold_wedge+10,32,SDL_HWSURFACE|SDL_DOUBLEBUF)
;
    if ( screen == NULL )
    {
      printf("Unable to set 640x480 video: %s\n", SDL_GetError());
      exit(1);
    }

//////////////////////
//SDL initialization//
//////////////////////
///////////////////////
//Declaring Variables//
///////////////////////

    short x;
    short y;
    short z;
    short done = 0;
    short time = 0;
    float width[number];
    float height[number];
    float weight[number];
    float squariness[number];
    float density[number];
    float COMx;
    float COMy;
    float COMerr;
    package last;
    SDL_Event event;

////////////////////////
//Declaring Variables//
////////////////////////
/////////////////////////
//Initializing Weights//
/////////////////////////
/////////////////////////
//Initializing Weights//
/////////////////////////
//////////////////////////
//Initializing Packages//
//////////////////////////

    for(x=0; x<number; ++x)
    {
      pkgs[x].setside1(20+x);
      pkgs[x].setside2(20+x);
      pkgs[x].setmass(150-x);
    }



//    pkgs[32].setside1(150);
//    pkgs[32].setside2(150);
//    pkgs[32].setmass(1);

//////////////////////////
//Initializing Packages//
//////////////////////////
//////////////////////////////
//Initializing Sorting Vars//
//////////////////////////////

    for(x = 0; x < number; ++x)
      {
        width[x] = (short)floor(pkgs[x].getSide2());
        height[x] = (short)floor(pkgs[x].getSide1());
        weight[x] = (short)floor(pkgs[x].getMass());
        squariness[x] = pkgs[x].getSquariness();
        density[x] = pkgs[x].getDensity();
      }

//////////////////////////////
//Initializing Sorting Vars//
//////////////////////////////
///////////////////////////////////////////////
//Sorting Packages, Drawing Hold, Sequencing//
///////////////////////////////////////////////

   weight_order = new short[number];
   size_order = new short[number];
   squariness_order = new short[number];
   density_order = new short[number];

   pkgs = new package[number];
   order = new package[number];
   make_pkgs(width, height, weight, squariness, density, weight_order, size_order,
squariness_order, density_order, number);

   for(x=0;x<number;++x)
     pkgs[x]=order[x];

/////////////////////////////////////////////
//Sorting Packages, Drawing Hold, Sequencing//
/////////////////////////////////////////////
//////////////////
//Printing Block//
//////////////////

   for(x = 0; x < number; ++x)
     cout << "Weight " << x << " = " << order[x].getMass() << endl;
     cout << "Weight Order:" << endl;
   for(x = 0; x < number; ++x)
     cout  << weight[weight_order[x]] << endl;
     cout << "Size Order:" << endl;
   for(x = 0; x < number; ++x)
     cout << width[size_order[x]]*height[size_order[x]] << endl;
     cout << "Squariness Order:" << endl;
   for(x = 0; x < number; ++x)
     cout << squariness[squariness_order[x]] << endl;
     cout << "Density Order:" << endl;
   for(x = 0; x < number; ++x)
     cout << density[density_order[x]] << endl;
     cout << endl;

//////////////////
//Printing Block//
//////////////////
//////////////////////////////////////////////////////////////////
//Flip screen, Declare input var, Initialize loading and drawing//
//////////////////////////////////////////////////////////////////

  node input1,input2,input3,input4,input5;

  input1.setgMass(rand() % 100);
  input1.setgArea(rand() % (100 - input1.getgMass()));
  input1.setgDens(rand() % (100 - (input1.getgMass() + input1.getgArea())));
  input1.setgSqrn(rand() % (100 - (input1.getgMass() + input1.getgArea() + input1.getgDens())));
  input1.setRandom(100 - (input1.getgMass() + input1.getgArea() + input1.getgDens() +
input1.getgSqrn()));

  input2.setgMass(rand() % 100);
  input2.setgArea(rand() % (100 - input2.getgMass()));
  input2.setgDens(rand() % (100 - (input2.getgMass() + input2.getgArea())));
  input2.setgSqrn(rand() % (100 - (input2.getgMass() + input2.getgArea() + input2.getgDens())));
  input2.setRandom(100 - (input2.getgMass() + input2.getgArea() + input2.getgDens() +
input2.getgSqrn()));

  input3.setgMass(rand() % 100);
  input3.setgArea(rand() % (100 - input3.getgMass()));
  input3.setgDens(rand() % (100 - (input3.getgMass() + input3.getgArea())));
  input3.setgSqrn(rand() % (100 - (input3.getgMass() + input3.getgArea() + input3.getgDens())));
  input3.setRandom(100 - (input3.getgMass() + input3.getgArea() + input3.getgDens() +
input3.getgSqrn()));

  input4.setgMass(rand() % 100);
  input4.setgArea(rand() % (100 - input4.getgMass()));
  input4.setgDens(rand() % (100 - (input4.getgMass() + input4.getgArea())));
  input4.setgSqrn(rand() % (100 - (input4.getgMass() + input4.getgArea() + input4.getgDens())));
  input4.setRandom(100 - (input4.getgMass() + input4.getgArea() + input4.getgDens() +
input4.getgSqrn()));

  input5.setgMass(rand() % 100);
  input5.setgArea(rand() % (100 - input5.getgMass()));
  input5.setgDens(rand() % (100 - (input5.getgMass() + input5.getgArea())));
  input5.setgSqrn(rand() % (100 - (input5.getgMass() + input5.getgArea() + input5.getgDens())));
  input5.setRandom(100 - (input5.getgMass() + input5.getgArea() + input5.getgDens() +
input5.getgSqrn()));

  som(input1,input2,input3,input4,input5);
   while(done == 0)
   {
     SDL_PollEvent(&event);
     ++time;
     SDL_Delay(10);
     priority_set(screen, pkgs, number, order, orderct, weight_order, size_order,
squariness_order, density_order);
 /*
     for(x=1; x<number; ++x)
     {
       if(pkgs[x].checkLoaded()==1)
       {
         draw_rect(screen, pkgs[x].getSide2(), pkgs[x].getSide1(), pkgs[x].getX()+5,
pkgs[x].getY()+5, 100, 75, 0);
         draw_rect(screen, pkgs[x].getSide2()-2, pkgs[x].getSide1()-2, pkgs[x].getX()+6,
pkgs[x].getY()+6, 200, 150, 0);
         for(y=0 ; y<x; ++y)
           DrawPixel(screen,(short)pkgs[x].getX()+y+5,(short)pkgs[x].getY()+y+5,0,0,0);
       }
       if(pkgs[x].checkLoaded()==2)
       {
         draw_rect(screen, pkgs[x].getSide2(), hold_wedge-pkgs[x].getY(), pkgs[x].getX()+5,
pkgs[x].getY()+5, 64, 0, 0);
         draw_rect(screen, pkgs[x].getSide2()-2, hold_wedge-pkgs[x].getY()-2, pkgs[x].getX()+6,
pkgs[x].getY()+6, 255, 0, 0);
       }

       if ( event.type == SDL_QUIT )  {  done = 1;  }
       if ( event.key.keysym.sym == SDLK_ESCAPE ) { done = 1; }

     }
   */
     if ( event.type == SDL_QUIT )  {  done = 1;  }
     if ( event.key.keysym.sym == SDLK_ESCAPE ) { done = 1; }



     SDL_Flip(screen);
   }

////////
//Draw//
////////

}
```

functions.cpp:

```
#include "functions.h"
#include <SDL.h>
#include <cmath>
#include <iostream>
#include "globalvars.h"
#include "package2d.h"
#include "som.h"

using namespace std;

int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

short fib(short num)
{
      if (num < 0) return -1;
      if (num == 0) return 0;
      if (num == 1) return 1;
      return fib(num - 1) + fib(num - 2);
}

void make_hold(float width, float height)
{
  hold_sedge = width;
  hold_wedge = height;
  hold_area = hold_sedge*hold_wedge;
}

void make_pkgs(float *width, float *height, float *weight, float *squariness, float *density,
short *weight_order, short *size_order, short *squariness_order, short *density_order, short
number)
{

  short x;
  short y;

  short number_temp;
  float size_temp;
  float weight_temp;
  float squariness_temp;
  float density_temp;

  float weight_copy[number];
  float width_copy[number];
  float height_copy[number];
  float squariness_copy[number];
  float density_copy[number];
  float area[number];

  short by_weight[number];
  short by_size[number];
  short by_beide[number];
  short by_squariness[number];
  short by_density[number];

  //First array is by_weight. We will start by copying the weight array into sometihng more
temporary.

  for(x = 0; x < number; ++x)  //copy the array
    weight_copy[x]=weight[x];

  weight_temp=-1;

  for(y = 0; y < number; ++y) //this one sets by_weight[y] to the position in weight[].
by_weight[0] is the heaviest one, by_weight[number] is the lightest one.
  {
    for(x = 0; x < number; ++x)  //this one runs thorugh the whoel array, finds the largest,
stores it's position, sets it to -1
    {
      if(weight_copy[x] > weight_temp)  //check to see if this one is larger than any previous
ones.
      {
        weight_temp = weight_copy[x];  //set the weight_temp: the current largest weight.
        number_temp = x;               //set the position of the weight_temp.
      }
    }
    weight_copy[number_temp] = -1;    //set the largest one to -1, so it can never be the largest
again.
    by_weight[y] = number_temp;  //plug the position into by_weight
    weight_temp = -1;  //reset for next loop
  }

  //this concludes arrangement by weight.

  for(x = 0; x < number; ++x)  //copy the array agian for use in determining by_beide
    weight_copy[x]=weight[x];
  //The next array is the by_size array. Ready go.

  for(x = 0; x < number; ++x)  //set up the area array for us in determining size.
    area[x]=width[x]*height[x];

  size_temp = -1; //same idea as weight_temp

  for(y = 0; y < number; ++y) //this one sets by_size[y] to the position in area[]. by_size[0] is
the largest one, by_size[number] is the smallest one.
  {
    for(x = 0; x < number; ++x)  //this one runs thorugh the whole array, finds the largest,
stores it's position, sets it to -1
    {
      if(area[x] > size_temp)  //check to see if this one is larger than any previous ones.
      {
        size_temp = area[x];  //set the size_temp: the current largest area.
        number_temp = x;               //set the position of the weight_temp.
      }
    }
    area[number_temp] = -1;    //set the largest one to -1, so it can never be the largest again.
    by_size[y] = number_temp;  //plug the position into by_size
    size_temp = -1;  //reset for next loop
  }

  //this conludes arrangement by size.
  //next by squariness

  for(x = 0; x < number; ++x)  //copy the array
    squariness_copy[x]=squariness[x];

  squariness_temp=-1;

  for(y = 0; y < number; ++y)
  {
    for(x = 0; x < number; ++x)
    {
      if(squariness_copy[x] > squariness_temp)
      {
        squariness_temp = squariness_copy[x];
        number_temp = x;
      }
    }
    squariness_copy[number_temp] = -1;
    by_squariness[y] = number_temp;
    squariness_temp = -1;
  }

  //and by density

  for(x = 0; x < number; ++x)  //copy the array
    density_copy[x]=density[x];

  density_temp=-1;

  for(y = 0; y < number; ++y)
  {
    for(x = 0; x < number; ++x)
    {
      if(density_copy[x] > density_temp)
      {
        density_temp = density_copy[x];
        number_temp = x;
      }
    }
    density_copy[number_temp] = -1;
    by_density[y] = number_temp;
    density_temp = -1;
  }

  //Time to copy the arrangements here into the actual arrays
  for(x = 0; x < number; ++x)  //copying the weights
    weight_order[x] = by_weight[x];
  for(x = 0; x < number; ++x)  //copying the size
    size_order[x] = by_size[x];
  for(x = 0; x < number; ++x)  //copying the squariness
    squariness_order[x] = by_squariness[x];
  for(x = 0; x < number; ++x)  //copying the density
    density_order[x] = by_density[x];

}

void load_package(short number, package *pkgs)
{

  package current;
  package last;
  short x;
  short y;
  short z;
  short fail=0;
  current = pkgs[number];
  last = pkgs[number-1];


  float lastx = last.getX();
  float lasty = last.getY();
  float lastw = last.getSide2();
  float lasth = last.getSide1();
  float currx = current.getX();
  float currw = current.getSide2();
  float currh = current.getSide1();

    if(lastx + lastw + currw <= hold_sedge)
      current.setlocation(lastx + lastw, 0);
    else
      current.setlocation(0,0);
  for(y=0;y<3;++y)
    for(x=0;x<number-1;++x)
    {
        for(z=0; z<current.getSide2(); ++z)
          if(current.getY()<pkgs[x].getY()+pkgs[x].getSide1())
            if(current.getX()+z>=pkgs[x].getX() &&
current.getX()+z<=pkgs[x].getX()+pkgs[x].getSide2())
              current.setlocation(current.getX(), pkgs[x].getY()+pkgs[x].getSide1());

     current.load();

     if(current.getY()+current.getSide1()>hold_wedge)
       fail = 1;

      if(fail==1)
        current.setunloadable();


    }

  pkgs[number] = current;

}




void draw_rect(SDL_Surface *screen,float width, float height, float xpos, float ypos,  short r,
short g, short b)
{

  short x;
  short y;
  short h = (short)height;
  short w = (short)width;
  short xsos = (short)xpos;
  short ysos = (short)ypos;

  Slock(screen);
  for(x = xsos; x < width + xsos; ++x)
    for(y = ysos; y < height + ysos; ++y)
      DrawPixel(screen, x, y, r, g, b);
  Sulock(screen);
}

void show(package *pkgslist, SDL_Surface *screen, short number, package *order)
{
   draw_rect(screen, hold_sedge+10, hold_wedge+10, 0, 0, 255,255,255);
   draw_rect(screen, hold_sedge, hold_wedge, 5, 5,90,90,90);

   SDL_Flip(screen);
   SDL_Event event;

   draw_rect(screen, pkgslist[0].getSide2(), pkgslist[0].getSide1(), pkgslist[0].getX()+5,
pkgslist[0].getY()+5, 100, 75, 0);
   draw_rect(screen, pkgslist[0].getSide2()-2, pkgslist[0].getSide1()-2, pkgslist[0].getX()+6,
pkgslist[0].getY()+6, 200, 150, 0);

//////////////////////////////////////////////////////////////////
//Flip screen, Declare input var, Initialize loading and drawing//
//////////////////////////////////////////////////////////////////
////////
//Load//
////////

   for(short x=0; x<number;++x)
   {
     pkgslist[x].checkLoaded();
     load_package(x, pkgslist);
   }

   for(short x=0; x<number; ++x)
     cout << "package number " << x << " located at (" << pkgslist[x].getX() << "," <<
pkgslist[x].getY() << ") and loading status = " << pkgslist[x].checkLoaded() << endl;

////////
//Load//
////////
//////////////////////////////
//Find and deal with the COMy//
//////////////////////////////
//////////////////////////////
//Find and deal with the COMy//
//////////////////////////////
////////
//Draw//
////////
   short x;
   short y;
   draw_rect(screen, hold_sedge+10, hold_wedge+10, 0, 0, 255,255,255);
   draw_rect(screen, hold_sedge, hold_wedge, 5, 5, 0,0,255);

       for(x=1; x<number; ++x)
       {
        if(pkgslist[x].checkLoaded()==1)
        {
         draw_rect(screen, pkgslist[x].getSide2(), pkgslist[x].getSide1(), pkgslist[x].getX()+5,
pkgslist[x].getY()+5, 0, 64, 0);
         draw_rect(screen, pkgslist[x].getSide2()-2, pkgslist[x].getSide1()-2,
pkgslist[x].getX()+6, pkgslist[x].getY()+6, 0, 255, 0);
        for(y=0;y<x;++y)
        DrawPixel(screen,(short)pkgslist[x].getX()+y+5,(short)pkgslist[x].getY()+y+5,0,0,0);
        }
        if(pkgslist[x].checkLoaded()==2)
        {
          draw_rect(screen, pkgslist[x].getSide2(), hold_wedge-pkgslist[x].getY(), 
pkgslist[x].getX()+5, pkgslist[x].getY()+5, 64, 0, 0);
          draw_rect(screen, pkgslist[x].getSide2()-2, hold_wedge-pkgslist[x].getY()-2, 
pkgslist[x].getX()+6, pkgslist[x].getY()+6, 255, 0, 0);
        }

      }
      short relevantY = (short)floor(nodeOfInterestPos.ypos);
      short relevantX = (short)floor(nodeOfInterestPos.xpos);
      float COMx = nodes[relevantY][relevantX].FindCOM(order, number).xpos;
      float COMy = nodes[relevantY][relevantX].FindCOM(order, number).ypos;
      short COMx_short = (short)floor(COMx);
      short COMy_short = (short)floor(COMy);
    Slock(screen);
     DrawPixel(screen, COMx_short, COMy_short, 255, 255, 255);
     DrawPixel(screen, COMx_short+1, COMy_short, 255, 255, 255);
     DrawPixel(screen, COMx_short+1, COMy_short+1, 255, 255, 255);
     DrawPixel(screen, COMx_short+1, COMy_short-1, 255, 255, 255);
     DrawPixel(screen, COMx_short-1, COMy_short, 255, 255, 255);
     DrawPixel(screen, COMx_short-1, COMy_short+1, 255, 255, 255);
     DrawPixel(screen, COMx_short-1, COMy_short-1, 255, 255, 255);
    Sulock(screen);
       SDL_Flip(screen);


}

void DrawPixel(SDL_Surface *screen, int x, int y, Uint8 R, Uint8 G, Uint8 B)
{
  Uint32 color = SDL_MapRGB(screen->format, R, G, B);
  switch (screen->format->BytesPerPixel)
  {
    case 1: // Assuming 8-bpp
      {
        Uint8 *bufp;
        bufp = (Uint8 *)screen->pixels + y*screen->pitch + x;
        *bufp = color;
      }
      break;
    case 2: // Probably 15-bpp or 16-bpp
      {
        Uint16 *bufp;
        bufp = (Uint16 *)screen->pixels + y*screen->pitch/2 + x;
        *bufp = color;
      }
      break;
    case 3: // Slow 24-bpp mode, usually not used
      {
        Uint8 *bufp;
        bufp = (Uint8 *)screen->pixels + y*screen->pitch + x * 3;
        if(SDL_BYTEORDER == SDL_LIL_ENDIAN)
        {
          bufp[0] = color;
          bufp[1] = color >> 8;
          bufp[2] = color >> 16;
        } else {
          bufp[2] = color;
          bufp[1] = color >> 8;
          bufp[0] = color >> 16;
        }
      }
      break;
    case 4: // Probably 32-bpp
      {
        Uint32 *bufp;
        bufp = (Uint32 *)screen->pixels + y*screen->pitch/4 + x;
        *bufp = color;
      }
      break;
  }
}
```

```
void Slock(SDL_Surface *screen)
{
  if ( SDL_MUSTLOCK(screen) )
  {
    if ( SDL_LockSurface(screen) < 0 )
    {
      return;
    }
  }
}

void Sulock(SDL_Surface *screen)
{
  if ( SDL_MUSTLOCK(screen) )
  {
    SDL_UnlockSurface(screen);
  }
}

/*void priority_set(node *nodes[])
{
     float fitnesses[MAXH*MAXW];
     float prevFitnesses[MAXH*MAXW];
     int fitinc = 0;
     for (int yinc = 0;yinc < MAXH;++yinc)
         for (int xinc = 0;xinc < MAXW;++xinc)
             {
                  nodes[yinc][xinc].setFitness(((1 / (float)nodes[yinc][xinc].getSteps())
                                           + (1 / nodes[yinc][xinc].hold_COM().xpos)
                                           + (1 / nodes[yinc][xinc].hold_COM().ypos)
                                            + ((float)nodes[yinc][xinc].loadedOK())) / 4);
                  prevFitnesses[fitinc] = fitnesses[fitinc];
                  fitnesses[fitinc] = nodes[yinc][xinc].getFitness();
                  ++fitinc;
             }
    qsort(fitnesses, MAXH*MAXW, sizeof(float), compare);

    int xlocs[3],ylocs[3],findage = 1;
    for (int yinc = 0;yinc < MAXH;++yinc)
      for (int xinc = 0;xinc < MAXW;++xinc)
        if (nodes[yinc][xinc].getFitness() == fitnesses[MAXH*MAXW - findage])
        {
         ylocs[findage - 1] = yinc;
         xlocs[findage - 1] = xinc;
         findage++;
        }
   if (fitnesses[MAXH*MAXW - 1] - prevFitnesses[MAXH*MAXW - 1] > IMP_TOLERANCE)
   {
    for (int i = 0;i < MAXH*MAXW;++i) prevFitnesses[i] = fitnesses[i];
    som(nodes[ylocs[2]][xlocs[2]], nodes[ylocs[1]][xlocs[1]], nodes[ylocs[0]][xlocs[0]]);
   }
   else
   {
       nodes[ylocs][2]][xlocs[2]].load_package();
       show();
   }
}*/



void priority_set(SDL_Surface *screen, package *pkgs, short number, package *order, short
orderct, short *weight_order, short *size_order, short *squariness_order, short *density_order)
{
     float fitnesses[MAXH*MAXW];
     float prevFitnesses[MAXH*MAXW];
     short weights[5];
     int fitinc = 0;
     for (int yinc = 0;yinc < MAXH;++yinc)
         for (int xinc = 0;xinc < MAXW;++xinc)
             {
                  nodes[yinc][xinc].FindCOM(pkgs, number);
                  nodes[yinc][xinc].setFitness(((1 / (float)nodes[yinc][xinc].getSteps())
                                             + (1 / abs(nodes[yinc][xinc].FindCOM(pkgs,
number).xpos - HOLD_CENTER_X))
                                             + (1 / abs(nodes[yinc][xinc].FindCOM(pkgs,
number).ypos - HOLD_CENTER_Y))
                                              + ((float)nodes[yinc][xinc].loadedOK())) / 4);
                  prevFitnesses[fitinc] = fitnesses[fitinc];
                  fitnesses[fitinc] = nodes[yinc][xinc].getFitness();
                  ++fitinc;
              }
    qsort(fitnesses, MAXH*MAXW, sizeof(float), compare);

    int xlocs[5],ylocs[5],findage = 1;
    for (int yinc = 0;yinc < MAXH;++yinc)
      for (int xinc = 0;xinc < MAXW;++xinc)
        if (nodes[yinc][xinc].getFitness() == fitnesses[MAXH*MAXW - findage])
        {
         ylocs[findage - 1] = yinc;
         xlocs[findage - 1] = xinc;
         findage++;
        }
   if (fitnesses[MAXH*MAXW - 1] - prevFitnesses[MAXH*MAXW - 1] > IMP_TOLERANCE)
   {
    for (int i = 0;i < MAXH*MAXW;++i) prevFitnesses[i] = fitnesses[i];
    som(nodes[ylocs[4]][xlocs[4]],nodes[ylocs[3]][xlocs[3]],nodes[ylocs[2]][xlocs[2]],
nodes[ylocs[1]][xlocs[1]], nodes[ylocs[0]][xlocs[0]]);
   }
   else
   {

       nodeOfInterestPos.ypos = ylocs[2];
       nodeOfInterestPos.xpos = xlocs[2];
       short relevantY = (short)floor(nodeOfInterestPos.ypos);
       short relevantX = (short)floor(nodeOfInterestPos.xpos);
       weights[0] = nodes[relevantY][relevantX].getgMass();
       weights[1] = nodes[relevantY][relevantX].getgArea();
       weights[2] = nodes[relevantY][relevantX].getgSqrn();
       weights[3] = nodes[relevantY][relevantX].getgDens();
       weights[4] = nodes[relevantY][relevantX].getrandom();


       nodes[(short)nodeOfInterestPos.ypos][(short)nodeOfInterestPos.xpos].sequence(pkgs, number,
order, orderct, weights, 5, weight_order, size_order, squariness_order, density_order);
       show(order, screen, number, order);
   }
}
```

functions.h:

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

//#include "package.h"
#include "globalvars.h"
#include "package2d.h"
#include "node.h"
#include <SDL.h>
#include <cmath>
#include <iostream>



int compare (const void * a, const void * b);

void make_hold(float width, float height);
void make_pkgs(float *width, float *height, float *weight, float *squariness, float *density,
short *weight_order, short *size_order, short *squariness_order, short *density_order, short
number);
void load_package(short number, package *pkgs);
void draw_rect(SDL_Surface *screen, float width, float height, float xpos, float ypos,  short r,
short g, short b);
void show(package *pkgslist, SDL_Surface *screen, short number, package *order);

void DrawPixel(SDL_Surface *screen, int x, int y, Uint8 R, Uint8 G, Uint8 B);
void Slock(SDL_Surface *screen);
void Sulock(SDL_Surface *screen);
void priority_set(SDL_Surface *screen, package *pkgs, short number, package *order, short
orderct, short *weight_order, short *size_order, short *squariness_order, short *density_order);


#endif
```

som.cpp:

```
#ifndef SOM_F
#define SOM_F

#include <stdlib.h>
#include <SDL.h>
#include <string>
#include <cmath>
#include <iostream>
#include "functions.h"
#include "globalvars.h"

#define DEFAULT_DEPTH 16
#define NUM_LEVELS 256
#define UPDATE(x) ((MAXW-3)/pow((double)(x+1.0),.1))+3
#define SEED 42
#define HOWLONG 3250//325*.4
#define SGN(x) ((x)>0 ? 1 : ((x)==0 ? 0:(-1)))
#define ABS(x) ((x)>0 ? (x) : (-x))
#define abs(x) ((x)>0 ? (x) : (-x))
#define LEARN(x)   (1.0/(x+4.25)+.75)
#define INPUTS 5

using namespace std;

static SDL_Surface *screen;

node BMU[INPUTS];
node input[INPUTS];
int timestamp;

int nWeights[4];

int fib(int num)
{
    if (num < 0) return -1;
    if (num == 0) return 0;
    if (num == 1) return 1;
    return fib(num - 1) + fib(num - 2);
}

void rweight_vectors()
{
 srand(SEED);
 for (int yinc = 0;yinc < MAXH;yinc++)
 {
      for (int xinc = 0;xinc < MAXW;xinc++)
      {
              nodes[yinc][xinc].setgMass(rand() % 100);
              nodes[yinc][xinc].setgArea(rand() % (100 - nodes[yinc][xinc].getgMass()));
              nodes[yinc][xinc].setgSqrn(rand() % (rand() % (100 - (nodes[yinc][xinc].getgMass()
+ nodes[yinc][xinc].getgArea()))));
              nodes[yinc][xinc].setgDens(rand() % (rand() % (100 - (nodes[yinc][xinc].getgMass()
+ nodes[yinc][xinc].getgArea() + nodes[yinc][xinc].getgSqrn()))));
```

```
              nodes[yinc][xinc].setRandom((rand() % (100 - (nodes[yinc][xinc].getgMass() +
nodes[yinc][xinc].getgArea() + nodes[yinc][xinc].getgSqrn() + nodes[yinc][xinc].getgDens())))); 

              nodes[yinc][xinc].xpos = xinc;
              nodes[yinc][xinc].ypos = yinc;
      }
 }
 for (int yinc = 0;yinc < MAXH;yinc++)
       for (int xinc = 0;xinc < MAXW;xinc++)
           if ((nodes[yinc][xinc].gMass
              + nodes[yinc][xinc].gArea
              + nodes[yinc][xinc].gSqrn
              + nodes[yinc][xinc].gDens
              + nodes[yinc][xinc].Random) > 100) {
                                        cout << "Error generating values.  Retrying." << endl;
                                       rweight_vectors();
            }
 return;
}

//************************************************************************
//UPDATE FUNCTION*********************************************************
//************************************************************************

void update()
{
 int gMass_ldiff  = 100;
 int gArea_ldiff  = 100;
 int gSqrn_ldiff  = 100;
 int gDens_ldiff  = 100;
 int Random_ldiff = 100;
 float dist;
 float update_distance = (double)UPDATE(timestamp);
 for (int whichInput = 0;whichInput < INPUTS;++whichInput)
 {
   gMass_ldiff  = 100;
   gArea_ldiff  = 100;
   gSqrn_ldiff  = 100;
   gDens_ldiff  = 100;
   Random_ldiff = 100;
  cout << endl << "Testing for input number " << whichInput << endl;
  //Find diff of BMU

  for (int yinc=0; yinc<MAXH; ++yinc)
  {
         for (int xinc=0; xinc<MAXW; ++xinc)
         {
     &n bsp;        if ((abs(nodes[yinc][xinc].gMass - input[whichInput].gMass) < gMass_ldiff)
                &&(abs(nodes[yinc][xinc].gMass - input[whichInput].gMass) > 0)
                &&(abs(nodes[yinc][xinc].gArea - input[whichInput].gArea) < gArea_ldiff)
                &&(abs(nodes[yinc][xinc].gArea - input[whichInput].gArea) > 0)
                &&(abs(nodes[yinc][xinc].gSqrn - input[whichInput].gSqrn) < gSqrn_ldiff)
                 &&(abs(nodes[yinc][xinc].gSqrn - input[whichInput].gSqrn) > 0)
                &&(abs(nodes[yinc][xinc].gDens - input[whichInput].gDens) < gDens_ldiff)
                &&(abs(nodes[yinc][xinc].gDens - input[whichInput].gDens) > 0)
                &&(abs(nodes[yinc][xinc].Random - input[whichInput].Random) < gDens_ldiff)
                &&(abs(nodes[yinc][xinc].Random - input[whichInput].Random) > 0))
                 {
   &nbs p;             gMass_ldiff = abs(nodes[yinc][xinc].gMass - input[whichInput].gMass);
                 gArea_ldiff = abs(nodes[yinc][xinc].gArea - input[whichInput].gArea);
                 gSqrn_ldiff = abs(nodes[yinc][xinc].gSqrn - input[whichInput].gSqrn);
                 gDens_ldiff = abs(nodes[yinc][xinc].gDens - input[whichInput].gDens);
                 Random_ldiff = abs(nodes[yinc][xinc].Random - input[whichInput].Random);
;
                 cout << "Found potential BMU for input number " << whichInput << endl;
                 }

         }
  }
  //find BMU

  for (int yinc = 0;yinc < MAXH;++yinc)
     for (int xinc = 0;xinc < MAXW;++xinc)
         if ((abs(nodes[yinc][xinc].gMass - input[whichInput].gMass) == gMass_ldiff)
          && (abs(nodes[yinc][xinc].gArea - input[whichInput].gArea) == gArea_ldiff)
          && (abs(nodes[yinc][xinc].gSqrn - input[whichInput].gSqrn) == gSqrn_ldiff)
          && (abs(nodes[yinc][xinc].gDens - input[whichInput].gDens) == gDens_ldiff)
          && (abs(nodes[yinc][xinc].Random - input[whichInput].Random) == Random_ldiff))
             {
         &n bsp;   BMU[whichInput] = nodes[yinc][xinc];
             cout << "BMU for input number " << whichInput << " at (" << xinc << "," << yinc <<
")" << endl;
             }

  //find nodes near BMU

  for (int yinc = 0;yinc < MAXH;++yinc)
     for (int xinc = 0;xinc < MAXW;++xinc)
        {
         dist = sqrt
               (
                  pow(
                  (double)abs(nodes[yinc][xinc].xpo s - BMU[whichInput].xpos),2.0
                     )
                  + pow(
                  (double)abs(nodes[yinc][xinc].ypos - BMU[whichInput].ypos),2.0
                       )
                  );

         if ( dist < update_distance) { nodes[yinc][xinc].okToUpdate =
                  (update_distance-dist)/update_dis tance; }
            else { nodes[yinc][xinc].okToUpdate = 0; }
        }

  cout << endl << "Extecuting 'The actual disgusting update formula thing'" << endl;
 //The actual disgusting update formula thing

  for (int yinc = 0;yinc < MAXH;yinc++)
     for (int xinc = 0;xinc < MAXW;xinc++)
         {
         nodes[yinc][xinc].oldTemp = nodes[yinc][xinc].gMass;
         nodes[yinc][xinc].gMass = int(floor(nodes[yinc][xinc].gMass
                                 + (nodes[yinc][xinc].okToUpdate
                                 * (LEARN((float)timestamp))
                      ;            * ABS((float)input[whichInput].gMass -
(float)nodes[yinc][xinc].gMass))
                                 ));

         if (nodes[yinc][xinc].gMass - nodes[yinc][xinc].oldTemp > 0) {
                                     nodes[yinc][xinc].gArea  -= floor((nodes[yinc][xinc].gMass -
oldTemp) / 3);
                              & nbsp;      nodes[yinc][xinc].gDens  -=
floor((nodes[yinc][xinc].gMass - oldTemp) / 3);
                                     nodes[yinc][xinc].Random -= floor((nodes[yinc][xinc].gMass -
oldTemp) / 3);
                                     }

         if(nodes[yinc][xinc].okToUpdate == 1)
             cout << "Modified pixel at (" << xinc << ","  << yinc << ") for Red" << endl;
   &n bsp;     if(nodes[yinc][xinc].gMass>100)
            nodes[yinc][xinc].gMass=100;

         }


  for (int yinc = 0;yinc < MAXH;yinc++)
     for (int xinc = 0;xinc < MAXW;xinc++)
         {
         nodes[yinc][xinc].oldTemp = nodes[yinc][xinc].gArea;
         nodes[yinc][xinc].gArea = int(floor(nodes[yinc][xinc].gArea
       &nb sp;                         + (nodes[yinc][xinc].okToUpdate
                                 * (LEARN((float)timestamp))
                                 * ABS((float)input[whichInput].gArea -
(float)nodes[yinc][xinc].gArea))
                                 ));
         if (nodes[yinc][xinc].gArea - nodes[yinc][xinc].oldTemp > 0) {
&nb sp;                                    nodes[yinc][xinc].gMass  -=
floor((nodes[yinc][xinc].gArea - oldTemp) / 4);
                                     nodes[yinc][xinc].gSqrn  -= floor((nodes[yinc][xinc].gArea -
oldTemp) / 4);
                                     nodes[yinc][xinc].gDens  -= floor((nodes[yinc][xinc].gArea -
oldTemp) / 4);
                                      nodes[yinc][xinc].Random -= floor((nodes[yinc][xinc].gArea
- oldTemp) / 4);
                                     }

         if(nodes[yinc][xinc].okToUpdate == 1)
             cout << "Modified pixel at (" << xinc << ","  << yinc << ") for Green" << endl;
         if(nodes[yinc][xinc].gArea>100)
            nodes[yinc][xinc].gArea=100;

         }
    for (int yinc = 0;yinc < MAXH;yinc++)
     for (int xinc = 0;xinc < MAXW;xinc++)
         {
         nodes[yinc][xinc].oldTemp = nodes[yinc][xinc].gSqrn;
         nodes[yinc][xinc].gArea = int(floor(nodes[yinc][xinc].gSqrn
                                 + (nodes[yinc][xinc].okToUpdate
                         ;         * (LEARN((float)timestamp))
                                 * ABS((float)input[whichInput].gSqrn -
(float)nodes[yinc][xinc].gSqrn))
                                 ));
         if (nodes[yinc][xinc].gArea - nodes[yinc][xinc].oldTemp > 0) {
                                     nodes[yinc][xinc].gMass  -= floor((nodes[yinc][xinc].gSqrn -
oldTemp) / 4);
    &nbs p;                                nodes[yinc][xinc].gArea  -=
floor((nodes[yinc][xinc].gSqrn - oldTemp) / 4);
                                     nodes[yinc][xinc].gDens  -= floor((nodes[yinc][xinc].gSqrn -
oldTemp) / 4);
                                     nodes[yinc][xinc].Random -= floor((nodes[yinc][xinc].gSqrn -
oldTemp) / 4);
                                      }

         if(nodes[yinc][xinc].okToUpdate == 1)
             cout << "Modified pixel at (" << xinc << ","  << yinc << ") for Green" << endl;
         if(nodes[yinc][xinc].gArea>100)
            nodes[yinc][xinc].gArea=100;

         }
                    &nb sp;
  for (int yinc = 0;yinc < MAXH;yinc++)
     for (int xinc = 0;xinc < MAXW;xinc++)
         {
         nodes[yinc][xinc].oldTemp = nodes[yinc][xinc].gDens;
         nodes[yinc][xinc].gDens = int(floor(nodes[yinc][xinc].gDens
                                 + (nodes[yinc][xinc].okToUpdate
                                 * (LEARN((float)timestamp))
                                  * ABS((float)input[whichInput].gDens -
(float)nodes[yinc][xinc].gDens))
                                 ));

         if (nodes[yinc][xinc].gDens - nodes[yinc][xinc].oldTemp > 0) {
                                     nodes[yinc][xinc].gMass  -= floor((nodes[yinc][xinc].gDens -
oldTemp) / 4);
                     &n bsp;               nodes[yinc][xinc].gArea  -=
floor((nodes[yinc][xinc].gDens - oldTemp) / 4);
                                     nodes[yinc][xinc].gSqrn  -= floor((nodes[yinc][xinc].gDens -
oldTemp) / 4);
                                     nodes[yinc][xinc].Random -= floor((nodes[yinc][xinc].gDens -
oldTemp) / 4);
```

```
                                 }
&nbs p;
         if(nodes[yinc][xinc].okToUpdate == 1)
             cout << "Modified pixel at (" << xinc << ","  << yinc << ") for Blue" << endl;
         if(nodes[yinc][xinc].gDens>100)
            nodes[yinc][xinc].gDens=100;

         }
  for (int yinc = 0;yinc < MAXH;yinc++)
     for (int xinc = 0;xinc < MAXW;xinc++)
         {
         nodes[yinc][xinc].oldTemp = nodes[yinc][xinc].Random
         nodes[yinc][xinc].Random = int(floor(nodes[yinc][xinc].Random
                                 + (nodes[yinc][xinc].okToUpdate
                                 * (LEARN((float)timestamp))
              &nb sp;                 * ABS((float)input[whichInput].Random -
(float)nodes[yinc][xinc].Random))
                                 ));

         if (nodes[yinc][xinc].Random - nodes[yinc][xinc].oldTemp > 0) {
                                     nodes[yinc][xinc].gMass  -= floor((nodes[yinc][xinc].Random
- oldTemp) / 3);
                        & nbsp;            nodes[yinc][xinc].gArea  -=
floor((nodes[yinc][xinc].Random - oldTemp) / 3);
                                     nodes[yinc][xinc].gSqrn  -= floor((nodes[yinc][xinc].Random
- oldTemp) / 4);
                                     nodes[yinc][xinc].gDens  -= floor((nodes[yinc][xinc].Random
- oldTemp) / 3);
                                     }
  & nbsp;
         if(nodes[yinc][xinc].okToUpdate == 1)
             cout << "Modified pixel at (" << xinc << ","  << yinc << ") for Blue" << endl;
         if(nodes[yinc][xinc].gDens>100)
            nodes[yinc][xinc].gDens=100;

         }
}


 return;
}


//****************************************************************************
//MAIN************************************************************************
//****************************************************************************



void som(node input1, node input2, node input3, node input4, node input5)
{
  int totals[INPUTS];

  input1.gMass = floor((learn()*input1.gMass + (1 - learn()*(rand() % 100)) / 2);
  input1.gArea = floor((learn()*input1.gArea + (1 - learn()*(rand() % 100)) / 2);
  input1.gSqrn = floor((learn()*input1.gSqrn + (1 - learn()*(rand() % 100)) / 2);
  input1.gDens = floor((learn()*input1.gDens + (1 - learn()*(rand() % 100)) / 2);
  input1.random = floor((learn()*input1.random + (1 - learn()*(rand() % 100)) / 2);

  totals[0] = input1.gMass + input1.gArea + input1.gSqrn + input1.gDens + input1.random;
  input1.gMass /= totals[0];
  input1.gArea /= totals[0];
  input1.gSqrn /= totals[0];
  input1.gDens /= totals[0];
  input1.random /= totals[0];

  input2.gMass = floor((learn()*input1.gMass + (1 - learn()*(rand() % 100)) / 2);
  input2.gArea = floor((learn()*input1.gArea + (1 - learn()*(rand() % 100)) / 2);
  input2.gSqrn = floor((learn()*input1.gSqrn + (1 - learn()*(rand() % 100)) / 2);
  input2.gDens = floor((learn()*input1.gDens + (1 - learn()*(rand() % 100)) / 2);
  input2.random = floor((learn()*input1.random + (1 - learn()*(rand() % 100)) / 2);
  totals[1] = input2.gMass + input2.gArea + input2.gSqrn + input2.gDens + input2.random;
  input2.gMass /= totals[1];
  input2.gArea /= totals[1];
  input2.gSqrn /= totals[1];
  input2.gDens /= totals[1];
  input2.random /= totals[1];

  input3.gMass = floor((learn()*input3.gMass + (1 - learn()*(rand() % 100)) / 2);
  input3.gArea = floor((learn()*input3.gArea + (1 - learn()*(rand() % 100)) / 2);
  input3.gSqrn = floor((learn()*input3.gSqrn + (1 - learn()*(rand() % 100)) / 2);
  input3.gDens = floor((learn()*input3.gDens + (1 - learn()*(rand() % 100)) / 2);
  input3.random = floor((learn()*input3.random + (1 - learn()*(rand() % 100)) / 2);

  totals[2] = input3.gMass + input3.gArea + input3.gSqrn + input3.gDens + input3.random;
  input3.gMass /= totals[2];
  input3.gArea /= totals[2];
  input3.gSqrn /= totals[2];
  input3.gDens /= totals[2];
  input3.random /= totals[2];

  input4.gMass = floor((learn()*input4.gMass + (1 - learn()*(rand() % 100)) / 2);
  input4.gArea = floor((learn()*input4.gArea + (1 - learn()*(rand() % 100)) / 2);
  input4.gSqrn = floor((learn()*input4.gSqrn + (1 - learn()*(rand() % 100)) / 2);
  input4.gDens = floor((learn()*input4.gDens + (1 - learn()*(rand() % 100)) / 2);
  input4.random = floor((learn()*input4.random + (1 - learn()*(rand() % 100)) / 2);

  totals[3] = input4.gMass + input4.gArea + input4.gSqrn + input4.gDens + input4.random;
  input4.gMass /= totals[3];
  input4.gArea /= totals[3];
  input4.gSqrn /= totals[3];
  input4.gDens /= totals[3];
  input4.random /= totals[3];

  input5.gMass = floor((learn()*input5.gMass + (1 - learn()*(rand() % 100)) / 2);
  input5.gArea = floor((learn()*input5.gArea + (1 - learn()*(rand() % 100)) / 2);
  input5.gSqrn = floor((learn()*input5.gSqrn + (1 - learn()*(rand() % 100)) / 2);
  input5.gDens = floor((learn()*input5.gDens + (1 - learn()*(rand() % 100)) / 2);
  input5.random = floor((learn()*input5.random + (1 - learn()*(rand() % 100)) / 2);

  totals[4] = input5.gMass + input5.gArea + input5.gSqrn + input5.gDens + input5.random;
  input5.gMass /= totals[4];
  input5.gArea /= totals[4];
  input5.gSqrn /= totals[4];
  input5.gDens /= totals[4];
  input5.random /= totals[4];



  while(timestamp<HOWLONG)
   update();

   for (int yinc = 0;yinc < MAXH;yinc++)
     for (int xinc = 0;xinc < MAXW;xinc++)
         {
              nWeights[0] = nodes[yinc][xinc].gMass;
              nWeights[1] = nodes[yinc][xinc].gArea;
              nWeights[2] = nodes[yinc][xinc].gSqrn;
              nWeights[3] = nodes[yinc][xinc].gDens;
              nWeights[4] = nodes[yinc][xinc].Random;

              for (int pkgCt;pkgCt < total_pkgs;pkgCt++)
               load(4, nWeights);


         }
}

#endif
```

som.h:

```
#ifndef SOM_H
#define SOM_H

void som(node input1, node input2, node input3, node input4, node input5);

#endif
```

node.cpp:

```
#include "node.h"
#include "package2d.h"
//#include "functions.h"
//#include "globalvars.h"
#include <iostream>

using namespace std;

node::node()
{
       xpos = 0;
       ypos = 0;
       gMass = 0;
       gArea = 0;
       gSqrn = 0;
       gDens = 0;
       random = 0;
       fitness = 0.0;
       timesteps = 0;
       numLoaded = 0;

       okToUpdate = 0;
}

unsigned int node::getXPos() { return xpos; }
unsigned int node::getYPos() { return ypos; }
unsigned int node::getgMass() { return gMass; }
unsigned int node::getgArea() { return gArea; }
unsigned int node::getgSqrn() { return gSqrn; }
unsigned int node::getgDens() { return gDens; }
unsigned int node::getrandom() { return random; }
unsigned int node::getOK()   { return okToUpdate; }
float node::getFitness() { return fitness; }
unsigned int node::getSteps() { return timesteps; }
unsigned int node::loadedOK() { return numLoaded; }

void node::setLoc(unsigned int x, unsigned int y)
{
 xpos = x;
 ypos = y;
}

void node::setgMass(unsigned int mass) { gMass = mass; }
void node::setgArea(unsigned int area) { gArea = area; }
void node::setgSqrn(unsigned int squariness) { gSqrn = squariness; }
void node::setgDens(unsigned int density) { gSqrn = density; }
void node::setRandom(unsigned int random_p) { random = random_p; }

void node::setOK(unsigned short value)
{
 okToUpdate = value;
}

void node::setFitness(float value)
{
     fitness = value;
}

void node::incSteps()
{
     ++timesteps;
}

void node::incLoaded()
{
     ++numLoaded;
}

short node::weighted_rand(short *weights, short weightct)
{
  short checkingarray[weightct+1];
  short x;
  short y;
  short temp;
  short temp2;
  short random;
  short maxnum;

  maxnum=0;
  for(x=0; x<weightct; ++x)
    maxnum+=weights[x];

//  srand(clock());
//  srand(0);
  random = rand() % maxnum+1;

  checkingarray[0] = 0;
  for(x=0; x<weightct; ++x)
    checkingarray[x+1] = checkingarray[x] + weights[x];

  cout << endl << random << endl << endl;

  for (x=0;x < weightct;++x) cout << x << " : " << checkingarray[x] << endl;

  for(x=0; x<weightct; ++x)
  {
    cout << "Is " << random << " between " << checkingarray[x] << " and " << checkingarray[x+1]
<< endl;
    if(random >= checkingarray[x] && random < checkingarray[x+1])
    {
      cout << endl << x+1 << endl;
      return x+1;
    }
  }

  return 0;
}


void node::sequence(package *pkgs, short pkgct, package *order, short orderct, short *weights,
       short weightnum, short *weight_order, short *size_order, short *squariness_order, short
      *density_order)
{
     short x;
     short y;
     short num;
     short next;
     short done;
     short clock = 1;
cout << "1 - enter" << endl << flush;
     for(x=0; x<pkgct; ++x)
     {
     clock*=2;

     next = weighted_rand(weights, weightnum);

cout << "Returned from weighted_rand." << endl;

     done = 0;
       while(done == 0)
       {
         switch(next)
         {
           case 1:
cout << "case 1:  entering " << next << endl << flush;
             num = weight_order[0];
             y=0;
cout << "case 1:  entering while loop " << num << endl << flush;
cout << "    Is it Loaded? " << num << " " << pkgs[num].checkLoaded() << endl << flush;
             while(pkgs[num].checkLoaded() == 1)
             {
               ++y;
               num = weight_order[y];
cout << "    Is it Loaded? " << num << " " << pkgs[num].checkLoaded() << endl << flush;
             }
cout << "case 1:  leaving while loop " << num << endl << flush;
             order[x] = pkgs[num];
             pkgs[num].load();
             done = 1;
cout << "case 1:  done = 1 Loop = " << x << " of " << pkgct << endl << flush;
             break;
           case 2:
             num = size_order[0];
             y=0;
             while(pkgs[num].checkLoaded() == 1)
             {
               ++y;
               num = size_order[y];
cout << "    Is it Loaded? " << num << " " << pkgs[num].checkLoaded() << endl << flush;
             }
             order[x] = pkgs[num];
             pkgs[num].load();
             done = 1;
cout << "case 2:  done = 1 Loop = " << x << " of " << pkgct << endl << flush;
             break;
           case 3:
             num = squariness_order[0];
             y=0;
             while(pkgs[num].checkLoaded() == 1)
             {
               ++y;
               num = squariness_order[y];
             }
             order[x] = pkgs[num];
             pkgs[num].load();
             done = 1;
cout << "case 3:  done = 1 Loop = " << x << " of " << pkgct << endl << flush;
             break;
           case 4:
             num = density_order[0];
             y=0;
             while(pkgs[num].checkLoaded() == 1)
             {
               ++y;
               num = density_order[y];
             }
             order[x] = pkgs[num];
             pkgs[num].load();
             done = 1;
cout << "case 4:  done = 1 Loop = " << x << " of " << pkgct << endl << flush;
             break;
            case 5:
              num = rand()%pkgct;
              while(pkgs[num].checkLoaded() == 1)
                num = rand()%pkgct;
             order[x] = pkgs[num];
             pkgs[num].load();
             done = 1;
cout << "case 5:  done = 1 Loop = " << x << " of " << pkgct << endl << flush;
             break;
           default:
             num = density_order[0];
             y=0;
             while(pkgs[num].checkLoaded() == 1)
             {
               ++y;
               num = density_order[y];
             }
             order[x] = pkgs[num];
             pkgs[num].load();
             done = 1;
cout << "default:  done = 1 Loop = " << x << " of " << pkgct << endl << flush;
             break;
         }
       }
     }
cout << "3 - exit" << endl << flush;
}


point node::FindCOM(package *pkgs, short pkgct)
{
     short x;
     float massSum=0;
     float multipleX=0;
     float multipleY=0;
     float distance;
     point COM;

     for(x=0; x<pkgct; ++x)
       if(pkgs[x].checkLoaded() == 1)
       {
         massSum+=pkgs[x].getMass();
         multipleX+=((pkgs[x].getSide2()/2+pkgs[x].getX()))*pkgs[x].getMass();
         multipleY+=((pkgs[x].getSide1()/2+pkgs[x].getY()))*pkgs[x].getMass();
       }

     COM.xpos = multipleX/massSum;
     COM.ypos = multipleY/massSum;

//     distance = sqrt((*COMx - HOLD_CENTER_X)*(*COMx - HOLD_CENTER_X)+(*COMy -
HOLD_CENTER_Y)*(*COMy - HOLD_CENTER_Y));
     return COM;
}
```

node.h

```
#ifndef NODE_H
#define NODE_H

//#include "globalvars.h"
#include "package2d.h"
#include "point.h"

class node{
public:
       node();
       unsigned int getXPos();
       unsigned int getYPos();
       unsigned int getgMass();
       unsigned int getgArea();
       unsigned int getgSqrn();
       unsigned int getgDens();
       unsigned int getrandom();
       unsigned int getOK();
       float getFitness();
       unsigned int getSteps();
       unsigned int loadedOK();

       void setLoc(unsigned int x, unsigned int y);
       void setgMass(unsigned int mass);
       void setgArea(unsigned int area);
       void setgSqrn(unsigned int squariness);
       void setgDens(unsigned int density);
       void setRandom(unsigned int random_p);
       void setOK(unsigned short value);
       void setFitness(float value);
       void incSteps();
       void incLoaded();

       void sequence(package *pkgs, short pkgct, package *order, short orderct, short *weights,
        short weightnum, short *weight_order, short *size_order, short *squariness_order, short
        *density_order);
       point FindCOM(package *pkgs, short pkgct);

private:
       unsigned int xpos;
       unsigned int ypos;
       unsigned int gMass,gArea,gSqrn,gDens,random;
       float fitness;
       unsigned int timesteps;
       unsigned int numLoaded;

       unsigned short okToUpdate;

       short weighted_rand(short *weights, short number);
       short weights[5];
};

#endif
```

package2d.cpp:

```
#include <iostream>
#include <string>
#include <cassert>
#include "package2d.h"

using namespace std;

package::package()
{
  side1 = 0.;
  side2 = 0.;
  squariness = 0.;
  mass = 0.;
  density = 0.;
  xpos = 0.;
  ypos = 0. ;
  loaded = 0;
}

//Set Functions.

void package::setside1(float sets1)
{
     side1 = sets1;
}
void package::setside2(float sets2)
{
     side2 = sets2;
}
void package::setmass(float setm)
{
     mass = setm;
}

void package::setlocation(float x, float y)
{
  xpos = x;
  ypos = y;
}
```

```
void package::load()
{
  loaded = 1;
}

void package::unload()
{
  loaded = 0;
}

void package::setunloadable()
{
  loaded = 2;
}

//Get Functions


float package::getSide1()
{

  return side1;

}

float package::getSide2()
{

  return side2;

}

float package::getMass()
{

  return mass;

}


float package::getDensity()
{

  return mass/(side1*side2);

}

float package::getArea()
{

  return side1*side2;

}

float package::getSquariness()
{
    float squariness;
    squariness = side1/side2;
    if(squariness >1)
      squariness = 1/squariness;
        return squariness;

}

float package::getX()
{
  return xpos;
}

float package::getY()
```

```
{
  return ypos;
}

short package::checkLoaded()
{
  return loaded;
}
```

package2d.h:

```
#ifndef PACKAGE2D_H
#define PACKAGE2D_H

#include <iostream>
#include <string>

class package
{
  public:
//constructor
    package();

    void setmass(float setm);
    void setside1(float sets1);
    void setside2(float sets2);
    void setlocation(float x, float y);
    void load();
    void unload();
    void setunloadable();

    float getMass();
    float getSide1();
    float getSide2();
    float getDensity();
    float getArea();
    float getWeightClass();
        float getSquariness();
        float getX();
        float getY();
        short checkLoaded();

  private:

    float side1;                        //one of the three sides.
    float side2;
    float squariness;
    float mass;                         //the mass of the package
    float density;
    float xpos;
    float ypos;
    short loaded;


};
#endif
```

point.cpp:

```
#include "point.h"

point::point()
{
              xpos = 0.0;
              ypos = 0.0;
}
```

point.h:

```
#ifndef POINT_H
#define POINT_H

class point {
public:
	point();
	float xpos;
	float ypos;
};

#endif
```

globalvars.cpp:

```
#include "globalvars.h"
//#include "package.h"

#define HOLD_CENTER_X 320
#define HOLD_CENTER_Y 240

#define center_px_x(a) (a + HOLD_CENTER_X)
#define center_px_y(a) (a + HOLD_CENTER_Y)

float hold_sedge; //The Slogo, part a
float hold_wedge;
float hold_area;
short total_pkgs;
short drawn_pkgs;

node nodes[MAXH][MAXW];

point nodeOfInterestPos;


/*
package *package_list;
package *large_pkgs;
package *heavy_pkgs;
package *square_pkgs; // In large_pkgs and heavy_pkgs, largest/heaviest is index 0
package largest_pkg;
package heaviest_pkg;
package smallest_pkg;
package *same_mass_pkgs[10];
package smallpkgs_by_mass[10];
*/
```

globalvars.h:

```
#ifndef GLOBALVARS_H
#define GLOBALVARS_H

#include "package2d.h"
#include "node.h"
#include "point.h"

#define HOLD_CENTER_X 320
#define HOLD_CENTER_Y 240
#define IMP_TOLERANCE 5
#define MAXH 10
#define MAXW 10
#define DEFAULT_WIDTH 10
#define DEFAULT_HEIGHT 10

#define center_px_x(a) (a + HOLD_CENTER_X)
#define center_px_y(a) (a + HOLD_CENTER_Y)

extern float hold_sedge;
extern float hold_wedge;
extern float hold_area;
extern short total_pkgs;
extern short drawn_pkgs;

extern node nodes[MAXH][MAXW];

extern point nodeOfInterestPos;


/*
extern package *package_list;
extern package *large_pkgs;
extern package *heavy_pkgs;
extern package *square_pkgs; // In large_pkgs and heavy_pkgs, largest/heaviest is index 0
extern package largest_pkg;
extern package heaviest_pkg;
extern package smallest_pkg;
extern package *same_mass_pkgs[10];
extern package smallpkgs_by_mass[10];
*/
#endif
```

References

1. Martello, Silvano and Toth, Paolo. (1990). Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons. ISBN 0-471-92420-2.

2. Michael R. Gary and David S. Johnson (1979).Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman. ISBN 0-7167-1045-5.

3. Kohonen, Teuvo. (2005) The Self-Organizing Map. Laboratory of Computer and Information Science, Adaptive Informatics Research Centre.

4. “NP-hard” From Wikipedia, the free encyclopedia - http://en.wikipedia.org/wiki/NP-hard

5. “Polynomial time” From Wikipedia, the free encyclopedia - http://en.wikipedia.org/wiki/Polynomial_time

6. “Non-deterministic Turing machine” From Wikipedia, the free encyclopedia - http://en.wikipedia.org/wiki/Non-deterministic_Turing_machine

7. “Knapsack problem” From Wikipedia, the free encyclopedia - http://en.wikipedia.org/wiki/Knapsack_problem

8. “Dynamic Programming” From Wikipedia, the free encyclopedia - http://en.wikipedia.org/wiki/Dynamic_programming

9. “Comp. Theory FAQ” – http://www.cs.uwaterloo.ca/~alopez-o/comp-faq/faq.html

10. Simple DirecMedia Layer – http://www.libsdl.org.

Many thanks to Tom Laub for providing a great deal of assistance in repairing our code.