# CS360 -- Lab 1 -- Chain Heal

- CS360 -- Systems Programming
- Written by Adam Disney, Spring, 2015.
- Abetted by James S. Plank
- This file: **http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/labs/Lab-1-Chain-Heal/index.html**
- Lab Directory: **~jplank/cs360/labs/Lab-1-Chain-Heal**

---

## What You Turn In, and How You Compile

You are to submit the file **chain_heal.c**, which the TA will compile as follows:

```
UNIX> gcc -Wall -Wextra -o chain_heal chain_heal.c -lm
UNIX>
```

---

## Introduction

For this lab, you are the mighty healing shaman Urgosa in the popular online role-playing game, Planet of Peacecraft (PoP). You and your companions work together to pacify ferocious monsters so that eventually you all co-exist on one happy and harmonious planet. Currently, however, the planet is far from peaceful. It is a battlefield of terror.

Your job as Urgosa is to keep your companions alive with your healing spell called *Chain Heal*. Chain Heal allows Urgosa to target a player on the battlefield and heal him/her. Afterward, the spell then jumps from that player to another player that is within a certain range and heals them as well. It continues to jump to subsequent players in this fashion. What does it mean to heal them? Well, every player has a both a current amount of pacification points (PP) and a maximum amount of PP. Chain Heal can restore PP up to the maximum amount for that player.

For example, suppose Chain Heal can restore 300PP and it hits a player with 100PP and a max of 500PP. That player will now have 400PP. Pretty simple, but if that player were to be healed again they would end up with 500PP, not 700PP.

Chain Heal has the following limitations:

- Urgosa can only cast the spell on targets within some **initial_range** (Urgosa may target himself as well).
- The range at which Chain Heal can jump from that initial target to any following targets is limited by some **jump_range**.
- Chain Heal can only heal a number of players up to some **num_jumps**.
- The potential amount of healing done to the initial target is equal to some **initial_power**.
- The potential amount of healing to subsequent targets is reduced by a factor of **power_reduction** every time it jumps.
- Chain Heal can only heal a player once.

Let's look at another example. Suppose Urgosa casts a Chain Heal with an **initial_power** of 500 and a **power_reduction** of 0.25 and it will go through the following people: (in this order)

- Adam_the_Warrior (100/500PP)
- Catherine_the_Great (400/450PP)
- Chad_the_Priest (45/400PP)
- James_the_Lightning_Lord (300/600PP)

Adam will be healed for 400 even though the spell has a power of 500 because he cannot be over his max PP. Now when the spell moves on to Catherine, the power will be reduced by 25% making the power 500 * (1 - 0.25) = 375. Again PP cannot exceed maximum so Catherine will be healed for 50. Now moving on to Chad, the power is now 375 * (1 - 0.25) = 281.25 so Chad will be healed for 281 (pacification points are integers so we round the power appropriately - use **rint()** from math.h, and remember to use "-lm" when you compile). Finally, the power becomes 281.25 * (1 - 0.25) = 210.9375 so James will gain 211PP.

---

## Your Job

You are to write **chain_heal.c** which finds the optimal path for Chain Heal (the most pacification points restored). It will be called as follows:

```
chain_heal initial_range jump_range num_jumps initial_power power_reduction < input_file
```

Each command line argument is an integer with the exception of **power_reduction**, which is a double. **Chain_heal** reads information about the players on the battlefield from stdin. Each line contains information about a single player and is composed of exactly 5 words:

1. **X** (Integer): The x-coordinate of the player on a a 2D grid.
2. **Y** (Integer): The y-coordinate of the player.
3. **Current_PP** (Integer): The player's current pacification points.
4. **Max_PP** (Integer): The player's maximum pacification points.

5. **Name** (String, max of 100 characters): The player's name.

One of the players will always be Urgosa_the_Healing_Shaman because he is the one casting the Chain Heal.

Here is the file for the earlier example, **small.txt**:

```
0 0 100 100 Urgosa_the_Healing_Shaman
2 0 100 500 Adam_the_Warrior
3 0 400 450 Catherine_the_Great
4 0 45 400 Chad_the_Priest
4 1 300 600 James_the_Lightning_Lord
```

You do not have to error-check the input. You may assume that it is in the correct format and that there are no two players with the exact same name.

Once you have determined the optimal path, you print the path, one player per line. Each line will have the name of the player healed followed by a space and the amount of healing done to them. In addition, you print a final line with the word "Total_Healing" followed by a space and the total amount of healing done by the Chain Heal.

Here's is an example of my output on **small.txt**:

```
UNIX> ./chain_heal 2 1 4 500 0.25 < small.txt
Adam_the_Warrior 400
Catherine_the_Great 50
Chad_the_Priest 281
James_the_Lightning_Lord 211
Total_Healing 942
```

---

## Constraints and Requirements

**X** and **Y** will be between -10000 and 10000. **Max_PP** will be between 1 and 10,000, and **Current_PP** will be between 0 and **Max_PP**. The name will be a maximum of 100 characters.

***You have to write this program in C*** in a single file, with no extra "helper" programs. That means no standard template library, no fields library and no libfdr library. I'll give some help below.

To make this explicit, you may have the following includes in your program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>           /* If you include this, you need to compile with -lm */
```

When you read standard input, go ahead and use **scanf()** to read words rather than lines. You can do this because we are guaranteeing that input is in the proper format.

You are not allowed to make any assumptions on the size of the input file (i.e. on the number of nodes in the graph).

---

## Algorithmic Strategy

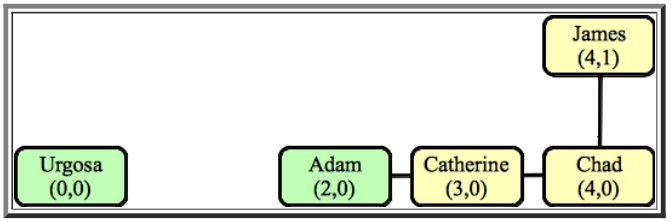
Clearly, this is going to be a graph problem. You read in all of the players, and create a node in the graph for each player. The graph is undirected, and there is an edge between two players if the distance between them is less than or equal to **jump_range**.

Once you create the graph, you determine each of the nodes that is within **initial_range** of Urgosa. For each of these nodes, you perform a depth-first search to find the optimal healing path starting from the node. The depth-first search enumerates all paths starting from the node. You maintain the best path, and at the end of the program, you print it.

For example, suppose I call:

```
UNIX> ./chain_heal 2 1 4 500 0.25 < small.txt
```

Here's a picture of the graph, with the potential starting nodes colored green:



There are only two potential paths for the Chain Heal:

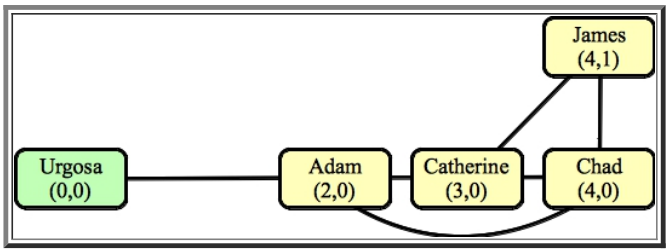
1. [Urgosa,0]
2. [Adam,400][Catherine,50][Chad,281][James,211]

Obviously, the second one has more healing.

Instead, suppose I call

```
UNIX> ./chain_heal 1 2 4 500 0.25 < small.txt
```

Now, the graph looks as follows:




There are now four possible 4-node paths:

1. [Urgosa,0][Adam,375][Catherine,50][Chad,211]
2. [Urgosa,0][Adam,375][Catherine,50][James,211]
3. [Urgosa,0][Adam,375][Chad,281][Catherine,50]
4. [Urgosa,0][Adam,375][Chad,281][James,211]

The last of these has the most healing: 867. Accordingly:

```
UNIX> ./chain_heal 1 2 4 500 0.25 < small.txt
Urgosa_the_Healing_Shaman 0
Adam_the_Warrior 375
Chad_the_Priest 281
James_the_Lightning_Lord 211
Total_Healing 867
UNIX>
```

---

## Implementational Strategy

As always, break this up into parts.

**Reading the input**: The first part should be simply reading the command line and reading the input file. Do this without storing anything, just so you can be sure your use of **scanf()** is correct.

**Creating the Nodes and putting them into an array**: As a second part, define a **Node** struct for each person. We're not going to worry about hooking the nodes together yet. We just want to read them in. As such, each node should have the following fields:

- **name**.
- **X** and **Y** coordinates.
- **Current_PP** and **Max_PP** values.
- A pointer to the previous node that you read.

Try something like the following:

```
typedef struct node
{
  char *name;
  int x, y;
  int cur_PP, max_PP;
  struct node *prev;
} Node;
```

To read in the nodes, go ahead and read the five words. Then allocate a new node with **malloc()**. Initialize the first five fields, and set **prev** to be the previous node that you read (obviously, with the first node, have its **prev** field be **NULL**). When you're done reading, you can traverse the nodes from most recent to least recent by chasing the **prev** pointers. What I did at this point was create an array of node pointers (Node **), and assigned nodes to it by traversing my linked nodes. I knew the size of this array, because its the number of nodes that I allocated (obviously, I kept track).

At this point, you no longer need the **prev** field in the nodes. You can traverse all of the nodes by traversing the array. This is some pretty old-school C programming, and you'll note how inconvenient it is compared to vectors in the STL. With vectors, you get that lovely **push_back()** method, which allows you to incrementally create vectors of arbitrary size. With C, you need to use a linked data

structure to read an arbitraty number of elements, and when you're done, you can allocate and create an array.

You should go ahead and test this program to make sure you're not making any mistakes. Then move onto:

**Creating the Graph**: I added two more fields to my nodes at this point:

```
int adj_size;
struct node **adj;
```

These define the adjacency lists for the nodes. I created the actual lists in three steps.

- In the first step, I calculated the size of the adjacency list for each node.
- In the second step, I allocated the adjacency list for each node. This is an array of node pointers, and I made it the *exact* size of the adjacency lists.
- In the third step, I put the nodes onto their adjacency lists.

Once again, this is where C is a pain compared to the STL. With the STL, I can simply use **push_back()** to put nodes onto adjacency lists. With C, I need to allocate the lists in their entirety before I put nodes onto them. I could have simply made the lists linked data structures (much like using **prev** when I read them in), too. You can do either -- both give you practice with C.

At this point, print out each node with its adjacency list and double-check yourself.

**Doing the DFS**: Now you should add a **visited** field to each node for your DFS. You should write a **DFS()** procedure, which has three arguments:

1. The node.
2. The hop number.
3. A pointer to a struct that contains global information (such as **num_jumps**, and **power_reduction**).

Now write a DFS that traverses all of the paths from each starting node. Have it print out [node,hop] for each time that it visits a node. Here are three examples, where you may want to make sure that your code matches mine (perhaps not the same order, but it should have the same visits):

```
UNIX> ./a.out 2 1 4 500 0.25 < small.txt
Node:Urgosa_the_Healing_Shaman  Hop 1
Node:Adam_the_Warrior  Hop 1
Node:Catherine_the_Great  Hop 2
Node:Chad_the_Priest  Hop 3
Node:James_the_Lightning_Lord  Hop 4
UNIX> ./a.out 1 2 4 500 0.25 < small.txt
Node:Urgosa_the_Healing_Shaman  Hop 1
Node:Adam_the_Warrior  Hop 2
Node:Catherine_the_Great  Hop 3
Node:Chad_the_Priest  Hop 4
Node:James_the_Lightning_Lord  Hop 4
Node:Chad_the_Priest  Hop 3
Node:Catherine_the_Great  Hop 4
Node:James_the_Lightning_Lord  Hop 4
UNIX> ./a.out 1 10 4 500 .25 < small.txt
Node:Urgosa_the_Healing_Shaman  Hop 1
Node:Adam_the_Warrior  Hop 2
Node:Catherine_the_Great  Hop 3
Node:Chad_the_Priest  Hop 4
Node:James_the_Lightning_Lord  Hop 4
Node:Chad_the_Priest  Hop 3
Node:Catherine_the_Great  Hop 4
Node:James_the_Lightning_Lord  Hop 4
Node:James_the_Lightning_Lord  Hop 3
Node:Catherine_the_Great  Hop 4
Node:Chad_the_Priest  Hop 4
Node:Catherine_the_Great  Hop 2
Node:Adam_the_Warrior  Hop 3
Node:Chad_the_Priest  Hop 4
Node:James_the_Lightning_Lord  Hop 4
Node:Chad_the_Priest  Hop 3
Node:Adam_the_Warrior  Hop 4
Node:James_the_Lightning_Lord  Hop 4
Node:James_the_Lightning_Lord  Hop 3
Node:Adam_the_Warrior  Hop 4
Node:Chad_the_Priest  Hop 4
Node:Chad_the_Priest  Hop 2
Node:Adam_the_Warrior  Hop 3
Node:Catherine_the_Great  Hop 4
Node:James_the_Lightning_Lord  Hop 4
Node:Catherine_the_Great  Hop 3
Node:Adam_the_Warrior  Hop 4
Node:James_the_Lightning_Lord  Hop 4
Node:James_the_Lightning_Lord  Hop 3
Node:Adam_the_Warrior  Hop 4
Node:Catherine_the_Great  Hop 4
```

```
Node:James_the_Lightning_Lord  Hop 2
Node:Adam_the_Warrior  Hop 3
Node:Catherine_the_Great  Hop 4
Node:Chad_the_Priest  Hop 4
Node:Catherine_the_Great  Hop 3
Node:Adam_the_Warrior  Hop 4
Node:Chad_the_Priest  Hop 4
Node:Chad_the_Priest  Hop 3
Node:Adam_the_Warrior  Hop 4
Node:Catherine_the_Great  Hop 4
UNIX>
```

**Calculating the total healing**: Now that you are convinced that you are enumerating all the paths, add a parameter **total_healing** to your **DFS()**, and a **best_healing** to your global information. Use this to calculate and store the best total healing for each path. Print it out at the end. Note that you're not maintaining the path at this point -- just the best total healing. Test it against the program in the lab directory:

```
UNIX> ./chain_heal 2 1 4 500 0.25 < small.txt | tail -n 1
Total_Healing 942
UNIX> ./chain_heal 1 2 4 500 0.25 < small.txt | tail -n 1
Total_Healing 867
UNIX> ./chain_heal 2 2 4 500 0.25 < small.txt | tail -n 1
Total_Healing 1086
UNIX> ./chain_heal 10 10 4 500 0.25 < small.txt | tail -n 1
Total_Healing 1086
UNIX> ./chain_heal 10 10 5 500 0.25 < small.txt | tail -n 1
Total_Healing 1086
UNIX> ./chain_heal 10 10 5 500 0.1 < small.txt | tail -n 1
Total_Healing 1105
UNIX> ./chain_heal 10 10 5 500 0.5 < small.txt | tail -n 1
Total_Healing 825
UNIX>
```

Finally, maintain the best path. What I did here was add three more fields to my global information:

```
int best_path_length;
Node **best_path;
int *healing;
```

Both **best_path** and **healing** are allocated to be **num_jumps** in size.

I also added a **healing** integer to each node. Finally, I added a **from** node to my DFS call. Now, during the DFS, each node maintains its current **healing** value, plus a pointer to the previous node in the path by reusing the **prev** field from above.

When a new "best" path is found, the nodes are stored in **best_path** (using the **prev** fields), and their healing values are stored in the **healing** array. The size of the path is stored in **best_path_length**. At the end of the program, these are used to print out the final best path.

---

## Couldn't This Be Faster, Dr. Plank?

Why yes, it could. You could make this a dynamic program by memoizing the DFS call. Don't do this, though, because you'd need something like the jrb library to implement the cache. (Or you could implement a hash table, but that's too much extra work). I'm just saying that you could make it faster with dynamic programming!

---

## Random Input Generation, Checking, Grading

The program **random_hero_gen.cpp** generates random input files. All of the gradescript examples were created with this program and the constraints above. The program seeds the random number generator on the current time, so it will generate different output in each second:

```
UNIX> ./random_hero_gen 10 10000 10000 10000
-8599   945  5234  8678 Urgosa_the_Healing_Shaman
-7118 -9489   429  3830 Spinny-McCrazyPants_the_Bladesinger
-1015  7421  1712  2300 Danielle_the_Gladiator
-2466  6917  1382  6477 Moon-Moon_the_Monk
 7429 -2081   691  7731 Hector_the_Delver
-2003  7930  3876  4071 Sabastian_the_Artificer
   60 -9294   703   760 Elizabeth_the_Shin-Kicker
  276  6332  1168  2066 Aldaricht_the_Hero,-Born-Under-Justice
-5935 -3947  3553  5955 Allen_the_Dragon-Knight
 7855 -2953   332  1746 Varrus_the_Sorcerer
UNIX> ./random_hero_gen 10 10000 10000 10000
-7075  7146   493  2890 Urgosa_the_Healing_Shaman
 3977 -2357  4144  6778 Chad_the_Rapper
 8477 -8470  5818  9776 Rob_the_Witchhunter
 9263  -161  6470  7377 Rock-Party_the_Cavalier
 6996  9754  2083  6215 Fist-RockBone_the_Smelly
```

```
-7449 -5631  2636  3179 Chad_the_Tiny
-4738  -273    41  4156 Chunkhead_the_Swashbuckler
 3570 -7102  1256  2998 Sabastian_the_Shaman
 1309  6582  2385  2439 Luigi_the_Adept
-6540  2149  6471  7318 Dylan_the_Grumpy-Pants
UNIX>
```

The program **chain_heal_check** can be used to check your output. It takes the same command line arguments as **chain_heal**, plus the name of the file used as standard input for **chain_heal**. On standard input, it takes the standard output of the **chain_heal** call. If standard input specifies a correct path (with the correct healings), then **chain_heal_check** simply exits. Otherwise, it specifies the error.

Note that **chain_heal_check** does *not* verify that the path is optimal. Just that it is legal.

```
UNIX> ./chain_heal 2 1 4 500 0.25 < small.txt | chain_heal_check 2 1 4 500 0.25 small.txt
UNIX> ./chain_heal 2 1 4 500 0.25 < small.txt | sed 's/James/Shmames/'
Adam_the_Warrior 400
Catherine_the_Great 50
Chad_the_Priest 281
Shmames_the_Lightning_Lord 211
Total_Healing 942
UNIX> ./chain_heal 2 1 4 500 0.25 < small.txt | sed 's/James/Shmames/' | chain_heal_check 2 1 4 500 0.25 small.txt
ERROR: Shmames_the_Lightning_Lord is not in the original input file
UNIX> ./chain_heal 2 1 4 500 0.25 < small.txt | sed 's/211/212/' | chain_heal_check 2 1 4 500 0.25 small.txt
ERROR: Incorrect healing amount on line 4
Read 212 but should be 211
UNIX>
```

The grading script works as in CS140 / CS302. If you are not familiar with these kinds of grading scripts, please talk with your TA's.

The way that the grading script works is that it calls the program in the lab directory to get the total healing value. Then it calls your program and checks to see if your total healing value is equal to the correct one. Then it calls **chain_heal_check** to make sure that your path is a correct one. As you'll note, your path does not have to equal mine -- it simply has to be legal and give the optimal total healing.