@CSPHILLI’S LEM-IN

HOW THE PROGRAM WORKS

The Core Data Structure

I use a dynamically sizing hash table in my project that can be altered at run time by using the ‘-l’ (ell) flag followed by an int value. For example, -l5 would set the load factor of the hash table to 5. The load factor is the theoretical capacity of each index when collisions occur. By having a value of 5, there are assumed to be at max of 5 entries at that particular index. Collisions occur when there are two identical primary identifiers. If you were storing names, “Joe” would have a specific index value and if you used another “Joe”, which was a different person, it would also be stored at the same index because a collision occurred.

Here is the output with the default load factor of 5:

```
HT[0]:
HT[1]: {Name: 1 | Links: 4,2,5}-->{Name: 10}
HT[2]: {Name: 2 | Links: 4,1,3,6}-->{Name: 11}
HT[3]: {Name: 3 | Links: 2,12}-->{Name: 12 | Comment: end | Links: 7,8,5,3}
HT[4]: {Name: 4 | Comment: start | Links: 1,2,13}-->{Name: 13 | Links: 4,14}
HT[5]: {Name: 5 | Links: 1,12}-->{Name: 14 | Links: 7,13}
HT[6]: {Name: 6 | Links: 8,2,9}
HT[7]: {Name: 7 | Links: 14,12}
HT[8]: {Name: 8 | Links: 12,6}
HT[9]: {Name: 9 | Links: 6}
```

And here it is using a load factor of just 1:

```
HT[0]:
HT[1]: {Name: 1 | Links: 4,2,5}-->{Name: 10}
HT[2]: {Name: 2 | Links: 4,1,3,6}-->{Name: 11}
HT[3]: {Name: 3 | Links: 2,12}-->{Name: 12 | Comment: end | Links: 7,8,5,3}
HT[4]: {Name: 4 | Comment: start | Links: 1,2,13}-->{Name: 13 | Links: 4,14}
HT[5]: {Name: 5 | Links: 1,12}-->{Name: 14 | Links: 7,13}
HT[6]: {Name: 6 | Links: 8,2,9}
```

```
HT[7]: {Name: 7 | Links: 14,12}
HT[8]: {Name: 8 | Links: 12,6}
HT[9]: {Name: 9 | Links: 6}
HT[10]:
HT[11]:
HT[12]:
HT[13]:
HT[14]:
HT[15]:
HT[16]:
HT[17]:
HT[18]:
HT[19]:
```

The load of a hash table is nbr_keys / current size of table. So the above would be 14 / 20 = 0.7. Each time a new key is added a check of load is performed. Once it exceeds the load factor, the hash table will double in size and reallocate.

As you can see, by decreasing the load factor, you're increasing the space complexity because you're reducing the amount of links each index can handle. And although in the above examples the load factor was reduced, the lists and their respective indices stayed the same due to the room names being numbers but the hash table doubled in size.

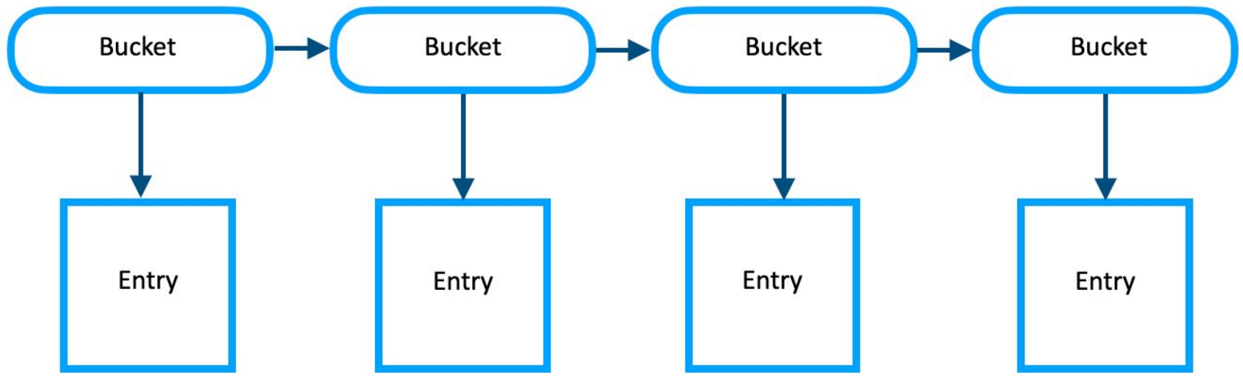
In terms of my project, I used the gen_key function which took a room's name and assigned an int value to it. Each character was added to each other and the resulting total was the key. Of course, if you had a room named "Joe" and another named "oeJ" the total would still be the same and a collision would occur. Despite there being identical key names, it's just used to get the index value of the hash table. Duplicate room names are not permitted in the program so it's okay to have the same key value because the room names will be unique.

Index Buckets & Entries

At each index is a linked list composed of buckets. These buckets have pointers to entries. Buckets also contain an important unsigned int called cap. Cap is used when performing the Edmonds-Karp algorithm later described in this document.

Entry structures hold all the necessary room information such as the key, x and y coordinates, links (which are pointers to other entries), visited status, a used flag (Edmonds-Karp), occ (short for occupied) and ant_id which are used for the printing algorithm.

To see that all easier, here is an image:

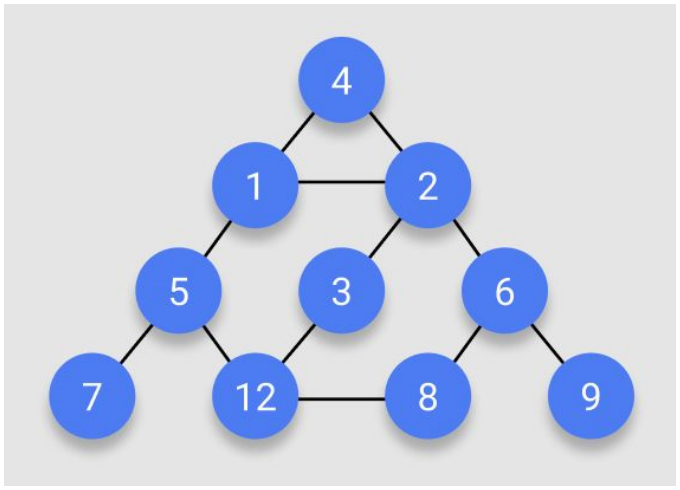


Edmonds-Karp & BFS

This program uses a similar variation of the Edmonds-Karp algorithm. The main difference is that in Lem-in, we have bi-directional flow and capacities limited to just 1. Each bucket contains the cap value which corresponds to the entry pointer in that respective bucket. Entry discovery is done via Breadth First Search and marking found paths is done using the EK method. On the following pages I'll illustrate how specifically a path is found, how the and how the nodes are altered after that.

Initial Map

Consider the following map which has a starting node of 4 and an end node of 12. I haven't colored them because my program checks for solutions both from the start and end since each can have a different amount of paths leading out of them. In this example, 4 has 2 paths leading out and 12 has 3. This example will just show you the steps involved with finding one path. The rest is just repetition.



The Main Steps

The main steps in this algorithm are:

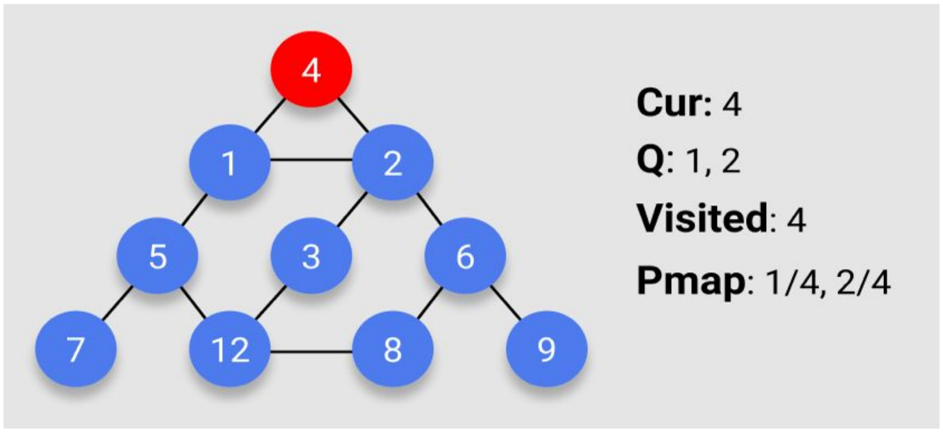
I. Add the starting node to the Q and mark it visited.
II. Set the *cur* variable to the value at the top of the Q.
III. Pop from the Q.
IV. Using the links of the *cur* entry, explore eligible links and add them to the *pmap.*
V. Repeat steps 1-4 until the end node is found.
VI. Create the path based on *pmap.*
VII. As the path is being created, mark the capacity to zero and set the node to *used.*
VIII. Clear the Q and pmap data, and start exploring the next option (if any).
IX. Rinse and repeat until BFS cannot search anything else.
X. Do steps 1-9 only begin with the end node.
XI. Choose the optimal solution set based on the results from ant distribution.
XII. Print out solution.

Steps In Detail

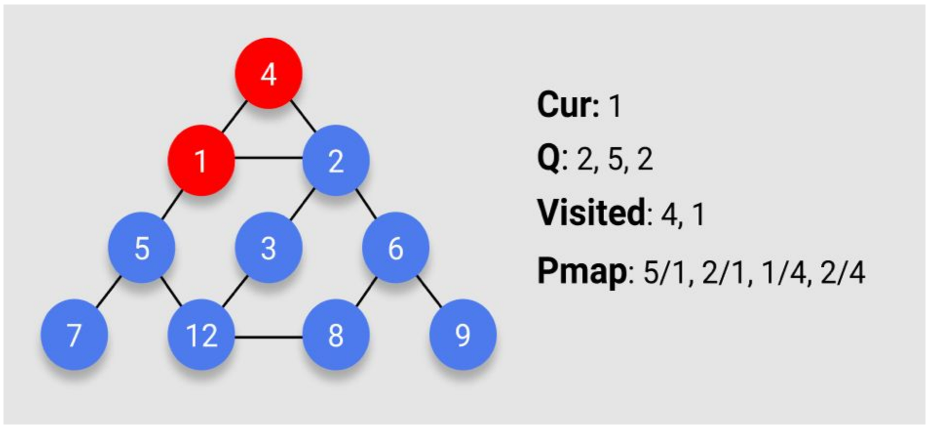
Since the above can be hard to "see", I'll go through the steps involved with finding the two paths from point 4.

We start by defining an int counter. This counter is equal to the amount of paths leading out of our start node. It will serve as the upper limit of our BFS loop.

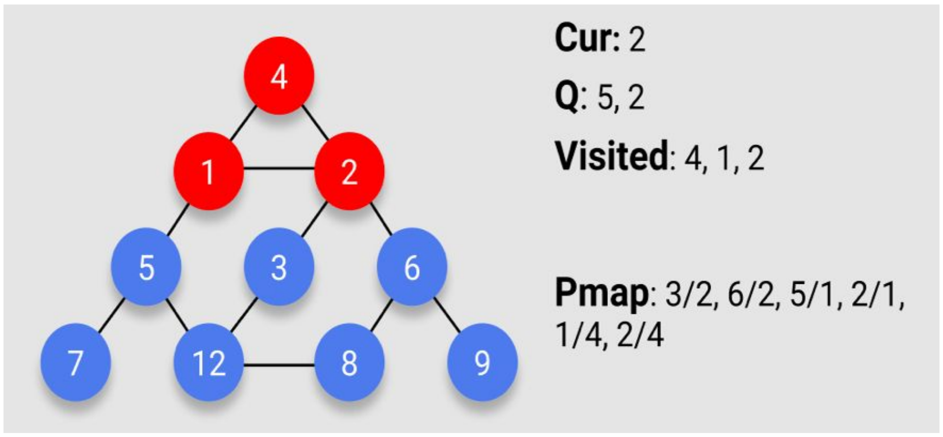
Starting at 4, set 4 to cur, pop 4 from Q, and add 4 to visited. Explore links of entry 4. When adding possible links to the Q, the links cannot be flagged as visited or used. Add the links to pmap. It's read like "1 was found via 4" and "2 was found via 4".



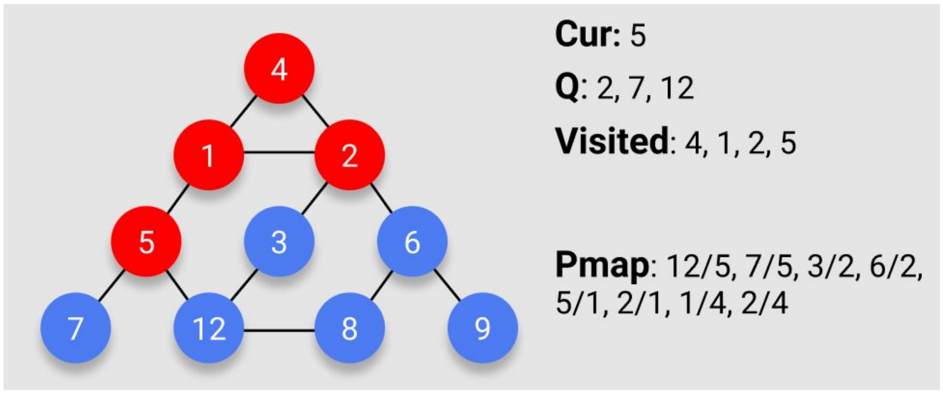
Next, set cur to 1, pop 1 from the Q, add 1 to visited and explore. It's important to note that duplicates can be found within the Q but they won't be explore because they have already been marked as visited.



2 is next in the Q so set cur equal to that, pop 2 from the Q, mark it visited, and then only add 3 and 6 to Q because although 1 and 4 are connections, they've been visited. The links are added to the pmap as well.

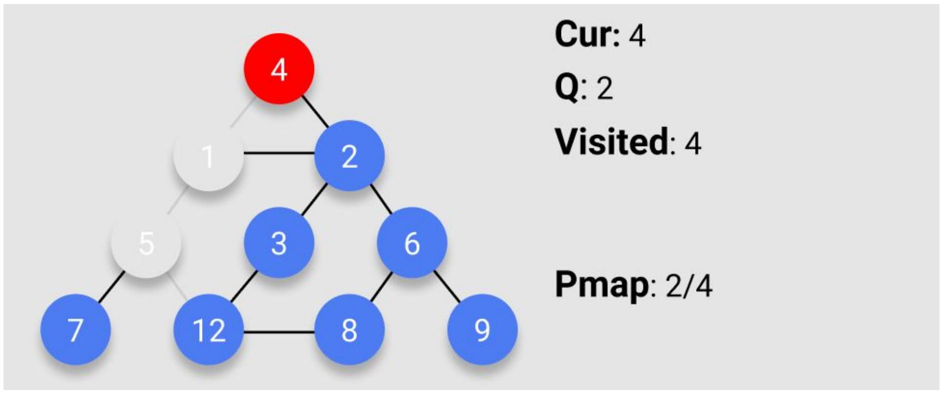


Next up is 5, so we set 5 as cur, pop it from the Q, mark it visited, and add 7 and 12 to the Q as well as populating the pmap. We found the end node.



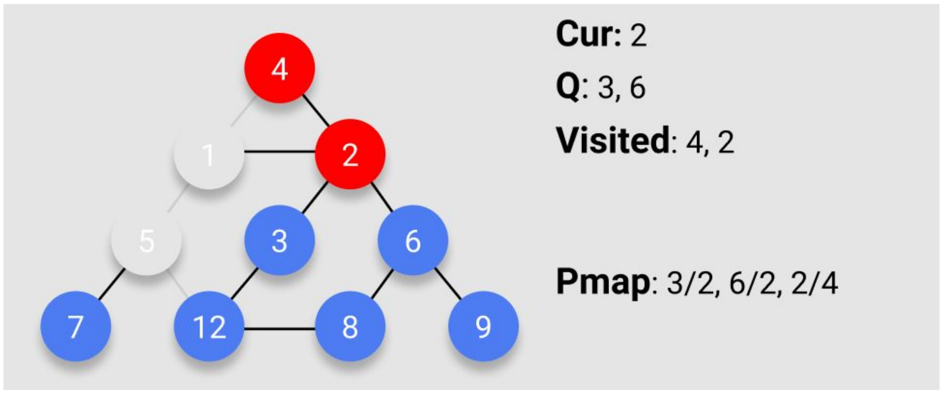
The PMAP

PMAP is short for position map. It is a linked list data structure. Each link has two entry pointers *fnd* and *via.* Fnd is short for found. The pmap is used to find the path back to the start node from the goal/end node. Since our end node is 12, that is set to the *fnd* pointer



and *via* is set to 5. 12 and 5 are also then added to a linked list. 5 is flagged as *used* but 12 is not since it is our goal node and needs to be discoverable. The capacity from 5->12 is also set to 0. Now the algorithm looks for the next *fnd* of 5 which is the previous *via.* 1 gets added to the list. Now it looks for 1, and that's via 4. Once the start node is reached, the loop breaks. At this point, pmap is cleared, the Q and visited is cleared, and the path we just found is marked off limits to the BFS algorithm and it's also added to a list of list (lol) data structure. We start fresh with 4 as *cur* and only add 2 to the Q. 1 cannot be explored because both the node is set to *used* as well as the *cap* between 4 and 1.

The next stop is setting *cur* to 2, popping it from the Q, adding it to visited, and exploring eligible links and adding those to the pmap.



By now you should be able to see what will happen next. We set *cur* to 3, pop 3 from the Q, mark it as visited, and explore the only available node: 12 which is the goal node.

Our 2nd and final path chosen from 4 is: 4-2-3-12. Of course there are other paths, but BFS will find the shortest paths.

In the BFS data structure, there are 2 lol pointers. *S2E* and *E2S.* These are short for start-to-end and end-to-start. Since we've started from the start, these lists of paths are saved to s2e. However, before the next step can occur, the paths need to be reversed.

We then have two paths in s2e:

1. 12-5-1-4
2. 12-3-2-4

The ants will be distributed between these paths based on the following description.

Ant Distribution

The next stage in this program is to distribute the ants in such a way that will result in the smallest amount of moves. In the previous step of finding paths, when they were added to the list of list structure, they were inserted in ascending order based on length. That is important for this step because it allows for a cascade effect in distribution.

It would take 1 ant 3 moves to go from start to finish in either of the above paths. And it would take 2 ants 4 moves. The formula is therefore *nbr_ants + (list_length - 2) = moves*. If we had to move 10 ants through the two paths above most efficiently, there would be a distribution of 5 and 5.

A. 5 + (4 - 2) = 7
B. 5 + (4 - 2) = 7

It's not a total of 14 moves since we can utilize two paths. If we have more than one path, the minimum number of moves to solve the map would be the path that has the largest amount of moves.

It's less clear when we have a path of 4 nodes and a path of 5 or even larger variance with many more paths. How would that distribution look? The first step is finding out the minimum number of moves to get 1 ant through each path.

1. 12-5-1-4
2. 12-8-6-2-4

A. 1 + (4 - 2) = 3 moves
B. 1 + (5 - 2) = 4 moves

Given this equation, we could also find how many ants we can use based on a desired number of moves.

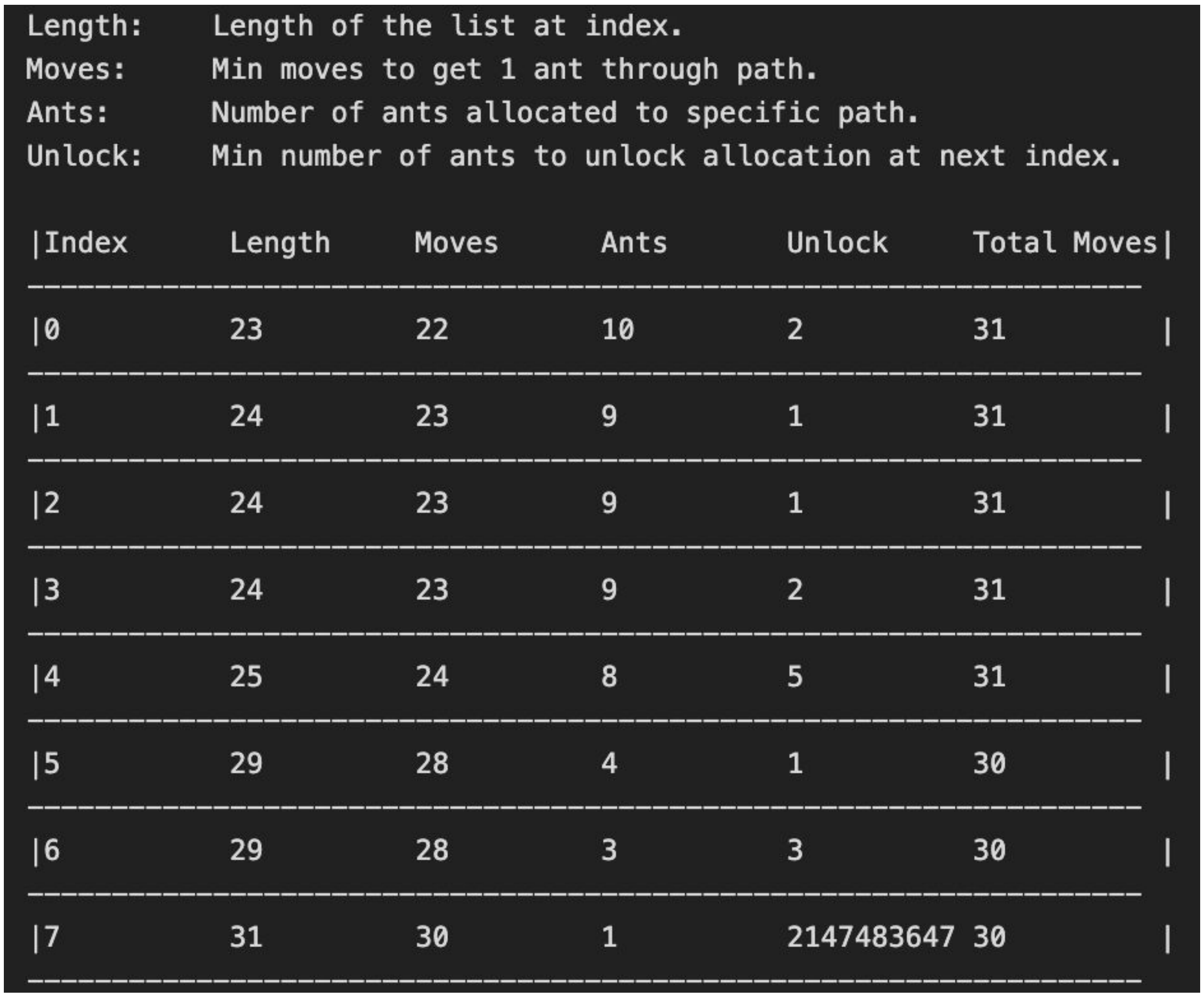
nbr_ants + (list_length - 2) = moves **becomes** *nbr_ants = moves - (list_length - 2)*

We want A to have the same moves as B: 4 - (4 - 2) = 2 ants. Therefore, path 1 can push 2 ants through in the same moves path 2 can only push 1 through. These baseline ant counts serve as the *unlock* gate in the distribution algorithm. The 2nd path won't have any allocation until path 1 has 2. And then the distribution can trickle into path 2. And it continues this way until all the ants have been allocated.

A distribution calculation is performed on both the s2e and e2s paths and the one with the lowest value is choses and assigned to the bfs->paths pointer. That is then sent to the final stage. An integer array of the ant distribution is also created and called *s_distro*. It helps in the creation of moves.

If this all sounds confusing, use flag -d and you'll see the distribution in action.

Here is a sample output of the distribution:



```
Length:    Length of the list at index.
Moves:     Min moves to get 1 ant through path.
Ants:      Number of ants allocated to specific path.
Unlock:    Min number of ants to unlock allocation at next index.

|Index      Length     Moves      Ants       Unlock     Total Moves|
-------------------------------------------------------------------
|0          23         22         10         2          31         |
-------------------------------------------------------------------
|1          24         23         9          1          31         |
-------------------------------------------------------------------
|2          24         23         9          1          31         |
-------------------------------------------------------------------
|3          24         23         9          2          31         |
-------------------------------------------------------------------
|4          25         24         8          5          31         |
-------------------------------------------------------------------
|5          29         28         4          1          30         |
-------------------------------------------------------------------
|6          29         28         3          3          30         |
-------------------------------------------------------------------
|7          31         30         1          2147483647 30         |
-------------------------------------------------------------------
```

Since index 7 has nothing else to compare to, it is set to the max int value.

Ants Marching

A reference to a song by the Dave Matthews Band, the last algorithm in this program is *ants_marching*. This function is in charge of assembling the moves and outputting the results in the fastest way possible.

Creating the Moves

Paths that are sent to *ants_marching* are sent in from the end to start. The algorithm will loop until *ants_e == max_ant.* Each path gets the following treatment in order:

1. Can an ant be moved into the end node? This is done by checking if the next node is occupied.
2. Now the algorithm parses through the path checking the next node if it's occupied. If so, it'll create the move and iterate forward again. It does this until the next entry is the start node.
3. If there is still an ant available in the respective *s_distro[index]* it'll subtract one from there and add the ant to the node.

Those steps will be repeated on each path until the list of paths expires. If the loop condition is still satisfied, it'll go through the paths again and do steps 1-3 for each.

Move Buffer

The actual moves are created using a series of ft_strcat functions and loaded into a buffer. The max size is 1025. Once the buffer has over 1000 chars, it will be dumped into a linked list structure, *t_output.* Output has a pointer to the line and the length of the line. If there are still more moves to be created, the buffer is cleared and the program continues.

White Spaces

At first, white spaces proved to be somewhat of a PITA. I was trying to figure out the rules when to add a whitespace to the end of a move. It became difficult when the algorithm was in the middle of the list of list, with no moves left for a particular path, but then the next path had moves. There wasn't really a guarantee of whitespace at the end of a move. It became a series of band aid fixes until I scrapped it and tried coming up with a better solution.

Then it dawned on me that every time there's a move, there is guaranteed to be a whitespace **preceding** the move. There's only one exception to that: When the move is the very first in the line. So I implemented a simple int value of i that toggled to 1 after the first move was done. Then each successive move checks if i == 1 and if so, a whitespace is inserted before the move. At the end of the list of list, the i value is reset to zero and a newline is appended to signal the end of the move.

Conclusion

This program proved to be quite the challenge on many levels. Patience was definitely a quality that was tested. You have the option to do this project as a group but I wanted to gain experience and knowledge in programming. I'm very glad that I did this even though it consumed a lot of time. I learned a lot from this. If you would've told me a year after I started programming that I would be writing a program that involves Edmonds-Karp, BFS, and dynamically sizing hash tables, I would've laughed.