**Abstract**

We present a multi-threaded pathfinding algorithm, implemented in clingo, to manage growing model size in answer set solutions.

While answer set programming pathfinding has been done before, the scope of these problems is either small, or their performance decays quickly with large data sets. Our program separates a large data set into smaller pieces, then solves each piece simultaneously, shifting agents and goals between pieces as needed. This methodology allows large maps to be traversed via multiple small, quick programs rather than one large, difficult to manage answer set. By keeping the answer sets small, we expand the size of problems able to be solved by answer set programming, as well as create a method for pathfinding that scales well even with the most demanding problem domains.

**Introduction**

Answer set programming is rarely used in problem sets that are not NP-hard. As answer set programming's ability to solve such problem sets is seen by most as it's most noticeable feature, practical search problems are seldom considered for answer set programming solutions. Our solver is an attempt to make a well scaling answer set pathfinding solution to large search spaces.

The search problem itself is comprised of a search space, made of vertices and edges, and agents and tasks. Each agent is given some number of tasks to complete, which may or may not be in it's portion of the search space. The search space is split into multiple pieces, referred to as servers, which will be searched concurrently.

The solver consists of two parts: one inner program, which manages the actual pathfinding, and a second outer one, which manages the information passed between inner programs. The inner program is written in clingo, and performs the actual pathfinding for it's portion of the search space. An instance of the inner program is run concurrently for each piece of the search space. It solves all goals within it's search space, and prepares all goals not found in it's search space to be passed to a different one. The outer program is written in python, and manages the inner programs, as well as passing them the necessary parameters in order to complete the goals of the search problem. When an inner program has marked an agent for transfer to a different search space, the outer program is used to move it to a neighboring one so it may continue it's search.

Information is passed from one server to another primarily through swap atoms. Swaps are created by the inner clingo solver whenever an agent's goal is not present in it's search space. This prompts the solver to move the agent to a vertice with an edge to a different search space, and to create a swap, an atom which contains all the information necessary to create a corresponding agent and goal in a neighboring server. The outer python function will then use this swap to make an agent and a goal, and add them to the server that the agents final vertice shared an edge with.

The outer program is also responsible for updating the locations of agents who are remaining in the server. This is done by finding the last move action taken by each agents, and updating it's starting position to its last move destination. The solvers are then run again, so the swapped agents can properly complete their goals.

**Implementation**

The inner portion of the program has a fairly simple implementation. It's implemented in clingo, specifically multi-shot clingo solving. Multi-shot clingo programs are split into a base, a

check, and a step portion. The parameters are an agent file, and a map file. The agent file consists of agents and tasks for them to perform. The map file consists of vertices and edges, which make up the search space of the program.

The base portion of the program is performed once, and sets up the initial information used in the solver. In our case, the base matches tasks to agents, forming goals for them to complete, and place our agents in their proper locations in the search space.

The check portion of the program is evaluated at the end of each step, and is used to check if the goal state is reached. Our goal state is when each goal has reached it's done state, which can be done in a variety of ways.

The step portion of the program is the core of multi-shot clingo solving. Every rule in the step portion of the program has a variable timestep, t, associated with it. The program then progressively runs the step portion, evaluating the check at the end of each run to see if the model is satisfied. There are only two actions in the program, move and stay. One action can be performed per agent per timestep. The action constraints are fairly straightforward. Agents can't be at two locations at once, one agent can't perform two actions in a time step, and two agents cannot move to the same location in one timestep. The goal of the step portion, as stated above is to get all the goals to a done state. There are two ways to do this. If the goal is present within the programs search space, a goal is considered done when the agent associated with it reaches the goal vertice. Alternatively, if the goal is not within the search space, the goal is considered done when the agent associated with it moves to a vertice that has an edge to a different server (search space). The goals include the server they are found in, and the program will prioritize edges that lead to servers that contain their goals. It then creates a "swap", an atom that contains all the information needed to recreate the agent and it's goal in a new server. These swaps are the primary way of passing information between different servers.

The outer portion of the program is implemented in python, and controls the inner, clingo, portions of the program. The outer portion takes a file as a parameter, which includes a

list of the agent and and map files the server programs will be using. The first thing the program does is create a thread for each server listed in the input file. Once the threads are completed, it captures their output and begins to process them so they can be used in the next iteration of the solver. This is accomplished through the use of several helper functions. The first simply cleans up the excess output generated by clingo. The next step the program takes is calling a helper method to find the last action taken by each of the agents in the server. This information is used to find the agent's new positions. Once the output of the first iteration of solvers has been processed, the program then writes the processed information to new files, which will serve as input for the next iteration of solvers. The program then runs the solvers again, and repeats the process until there are no swap atoms remaining, indicating that each agent has fulfilled all of its goals. Running the outer program is done like so:

python pythonshell.py input.txt  Where
input.txt is your input file.

**Solver Specifics**

The clingo solver has a few important atoms that help define the program. The search space is made of vertices and edges. Vertices are of the form:

vertice(X, Y): where X is the server number, and Y is the vertice number

Edges are of the form:

edge(X, Y, Z): where X is the server the edge leads to, Y is the starting vertice,
And Z is the destination vertice

These two atoms make up the map files, and define the search spaces. The biggest point of interest is the addition of the destination server on edges, which is used to help agents that are switching servers navigate.

The agents file is made of agents and tasks, which have the following forms:
agent(W, X, Y, Z): where W is the agent ID, X is the server the agent is starting in, Y is the agent starting vertice, and Z is the agent type (not used)

task(U, V, W, X, Y, Z): where U is the task ID, V is the server ID, W is the goal server, X is the group, Y is the goal vertice, and Z is the type (not used)

These two atoms make up the entirety of the agent files. The tasks will be assigned to agents and converted into goals - essentially tasks tied to an agent that must be completed.

The major atoms of the solver are swap and goal, and have following forms:
swap(R, S, V, U, T, W, X, Y, Z): where R is the task ID, S is the server, V is the agent ID, U is the goal, T is the group, W is the is the location , X is the server we're swapping to, Y is the goal server , and Z is the time step we created the swap.

goal(U, V, W, X, Y, Z): where U is the task ID, V is the server, W is the agent ID, X is The goal, Y is the group, and Z is the goal server

Out of all of these atoms swap is the most central to our program, and encapsulates all the information needed to make an agent and goal in a new server.

**Controller Specifics**

The basic layout of the controller program is as follows:

```
Main(input):
        For each server:
                Create clingo subprocess

        For each subprocess
        process.wait()

   For each subprocess:
      Clean up output

   For each server:
      new agents = updateagents()
Results_list[i] = updateswap()

   For each server:
      Create new agent file
```

Write to agent file

For each server:
Create clingo subprocess

For each subprocess:
process.wait()

The following are a list of all the helper functions that manage the solver output:

find_between(s, first, last):
Parameters: s is a string, first and last are chars
Outputs: a substring between the first and last chars given
Finds all contents of a string between two given chars, searching forward

find_between_r(s, first, last):
Parameters: s is a string, first and last are chars
Outputs: a substring between the first and last chars given
Finds all contents of a string between two given chars, searching backwards

processswap(output):
Parameters: output is a string, specifically an output string of our clingo solver
Outputs: a two element list. Element 0 is the processed swap, element 1 is the server
The swap is going to.
Processswap finds the first swap in the output parameter and turns it into an agent and
Goal atom. It then adds these atoms to a string, and returns this string, as well
As a server number. This will be used to add the processed swap to the proper Server
later in the program.

findserver(output)
Parameters: output is a string, specifically an output string of our clingo solver
Outputs: the server number of the output
Findserver is used to the server we are working with given the server output. It is used
To find the server parameter in updateagents.

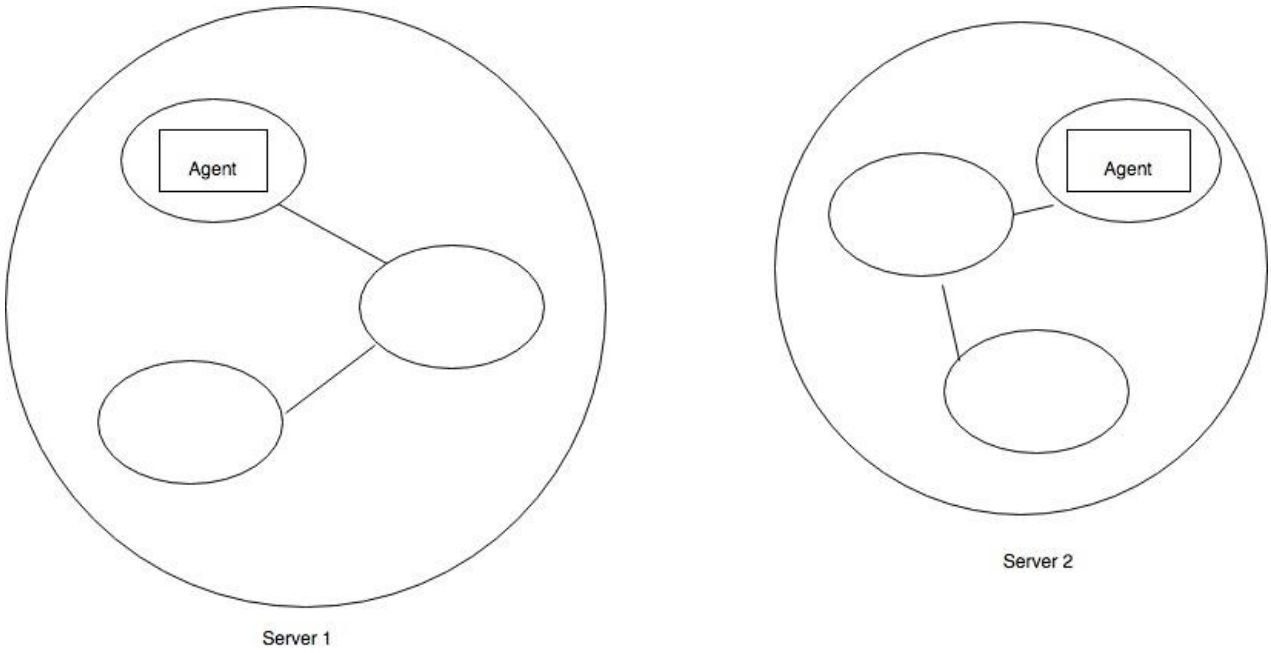
updateagents(ouput, numagents, server)
Parameters: output is a string, specifically an output string of our clingo solver.
Numagents is the number of agents in the server. The server is the server
Server number we are working with.
Output: returns a string containing the agents of the server, updated with their current
positions.
Updateagents is used to take a server, and to update the agents in it with their new
Positions. In order to do this, it takes the output of a clingo solver and searches for the
Last move action of each of the agents. The agents are then rewritten with their new
Positions, and added to our return string.

numagentsm(agentfile)
Parameters: agentfile is the name of an agenfile being used in the solver program
Outputs: the number of agents in the agent file
Numagents is used to count the number of agents in a given server. This allows us to Know how many agents are in need of being updated.

updateswap(output, agentlist, servers, i)
Parameters: output is a string, specifically an output string of our clingo solver.
Agentlist is a list of of agents in the server, updated to their current
Positions. Servers is the number of servers being used in the solver.
I is the server we are currently working on, starting at 0.
Outputs: returns a list of size servers, with the new agent list for each server in it
Updateswap is the next step after updateagents; it uses processswap to convert the
Swaps into agents and goals, then properly places them into their associated servers.
It also properly removes agents who have moved servers from their old ones. The end
Result is a list of agents ready to be used as input for the next round of solving.

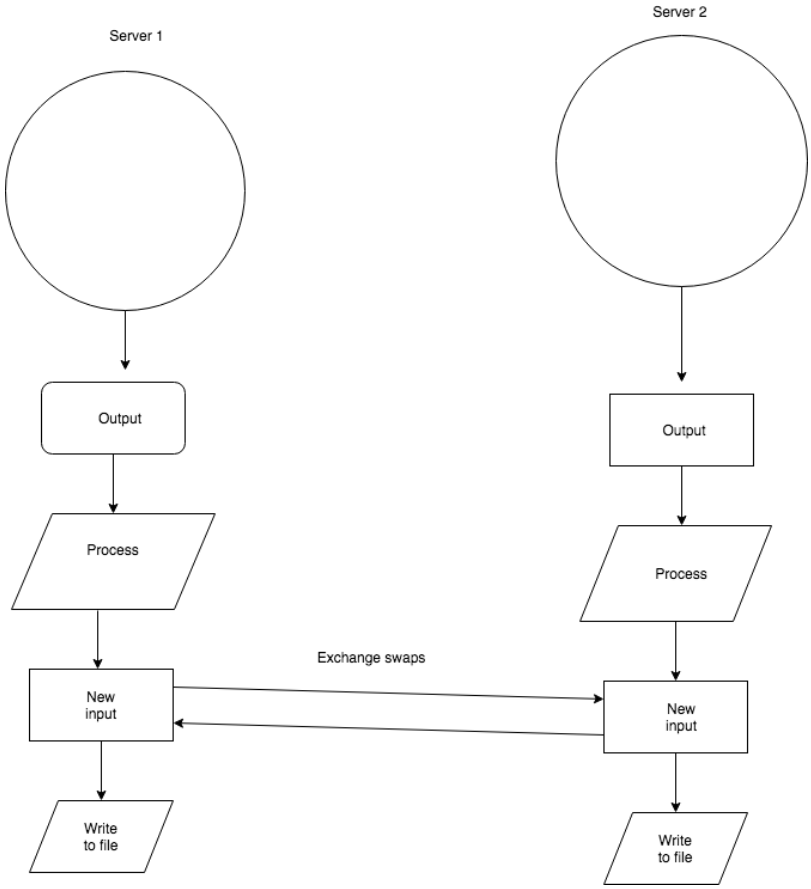
The flow of data for the program works like the following diagrams:

Step 1:



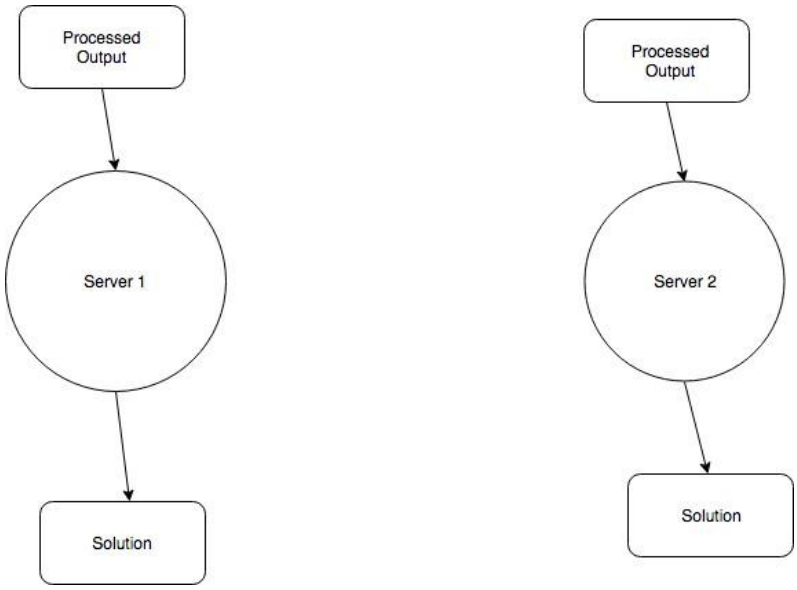

Individual servers solve their goals

Step 2:




The data is processed and swapped.

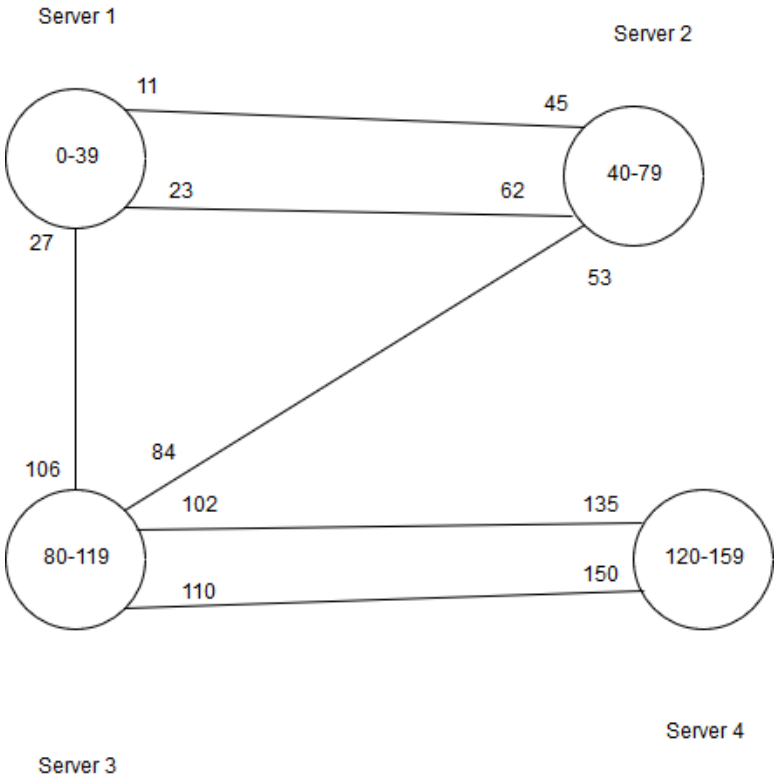
Step 3:




The processed data is used as input for the next round of solving.

**Test Results**

Our first test case is a 160 vertice map, which is visualized as follows:




. Each server has 40 vertices, 3 agents, and 6 tasks.

We split the map into 1, 2, and 4 servers, with the following results:

1 server: 5.585s
2 servers: 48.551s
4 servers: 5.25s

For the 2 server run we combine servers 1 and 2 as server 1 and server 3 and 4 as server 2. Each has 80 vertices, 6 agents, and 12 tasks.

These results are interesting, as the two server run is much much worse than simply using one server. Four servers, on the other hand have a marginal increase in performance over one. This is indicative of the need for larger test cases, as the two server run is probably a result of adding overhead without a huge performance increase as the servers are too small to be benefitted from splitting. The third run, however, is promising, as even with the addition of overhead, we see some performance boost over a single server map.

**Limitations:**

Our program has quite a few limitations. The first, and most important, is due to the lack of a visited list. Without a way to pass a visited list between servers, we don't have a way to remember which servers have been explored by a given agent. Due to this, all goals must be at the most within two jumps of a given server. This could limit the number of servers that can be used in larger test cases. Another limitation of the program is that each server must be strongly connected. If they aren't, there's a chance that a goal won't be solvable in its individual server, which will prevent the other servers from reaching the second solver phase. This is in itself a limitation of the program: if one of the agents is unable to complete its task, the entire program will be unable to complete.

**Future Work**

There are many interesting areas to explore with this work in the future. The most important remaining work is to change the program to be truly parallel. In the current implementation we run a round of clingo solvers in parallel, process the output, and then run another round of solvers. Ideally, we would simply pause when a server needed it's output processed, swap the agents accordingly, and continue, removing the need for multiple rounds of solver runs. I believe through the use of the Python clingo API that this is possible, although it would take a significant change in the way the iterative portions of the program are called, the default inc-mode of the clingo program would need to be modified as well. Another consideration is the current system limitation of not having a visited list. In conventional pathfinding of this nature, a visited list is used to ensure nodes (or in our case, servers) are not revisited, leading to an endlessly

searching agent. Unfortunately, in clingo, there is no way to pass a visited list between servers as an atom. This leads to the unfortunate limitation that each server can be at most two servers away from each other. Without this limitation, an agent with a goal over two servers away would end up searching aimlessly, with no way to tell whether any given server has already been searched or not. In a similar vein to this is the issue of collision over servers. As is, if two agents are swapping servers over the same edge, there is no way to make one wait until the edge is clear, which would lead to collisions in real world applications. This can hopefully be solved by moving to a truly parallel implementation, but as is it is a known issue. Some final implementation ideas would be to define different types of agents, only capable of performing specific tasks of their preferred type. This would simulate an automated factory or warehouse where different tasks would have to be completed by a certain type of robot, for example. It is my hope that once these issues are solved, this program can have many applications in pathfinding in large data sets.