Bio4CompSimpy User Manual

**Goals:**

There are two goals to this manual

1. Helping end users understand how to interact with the system and create simulations for different Bio4Comp biological devices

2. Explaining the code structure to programmers who wish to expend it/ improve it.

**Introduction**

Bio4CompSimpy is a simulator of Bio4Comp devices, as well as a framework for building custom such simulators. The simulation is done using the Beacon Calculus language and simulator engine, made by the Boemo Group.

For further details about BC language, please refer to:

<https://www.boemogroup.org/the-beacon-calculus>

<https://github.com/MBoemo/bcs>

**Installation:**

1. Install BCS as instructed in MBoemo's GitHub repo.

2. Download the Bio4CompSimpy folder from our GitHub repository: <https://github.com/ItamarDayan/Bio4CompSimpy>, and place it inside your beacon-calculus folder.

\* The program was written for use on Unix based operating systems only

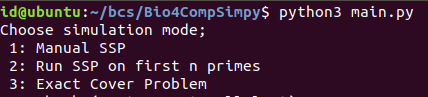
1. END USER MANUAL:

To start Bio4CompSimpy navigate in the terminal to the working directory of Bio4CompSimpy and then run main.py

(using the following command: python3 main.py).

Note that the code was written in python3 so running it on older versions probably will not work.

Running main.py will show you the main menu:



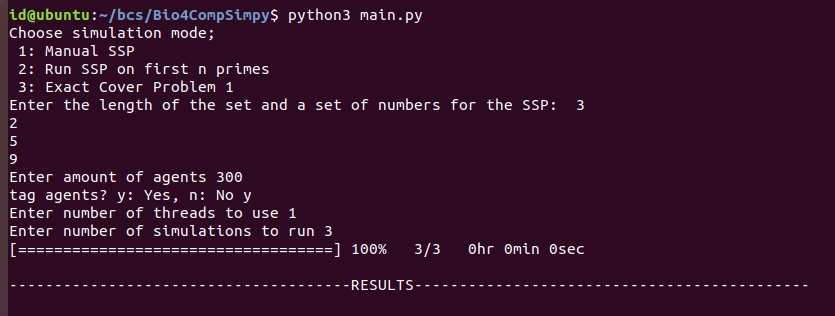
Here you can choose the problem you want to solve using the B4CMP simulator

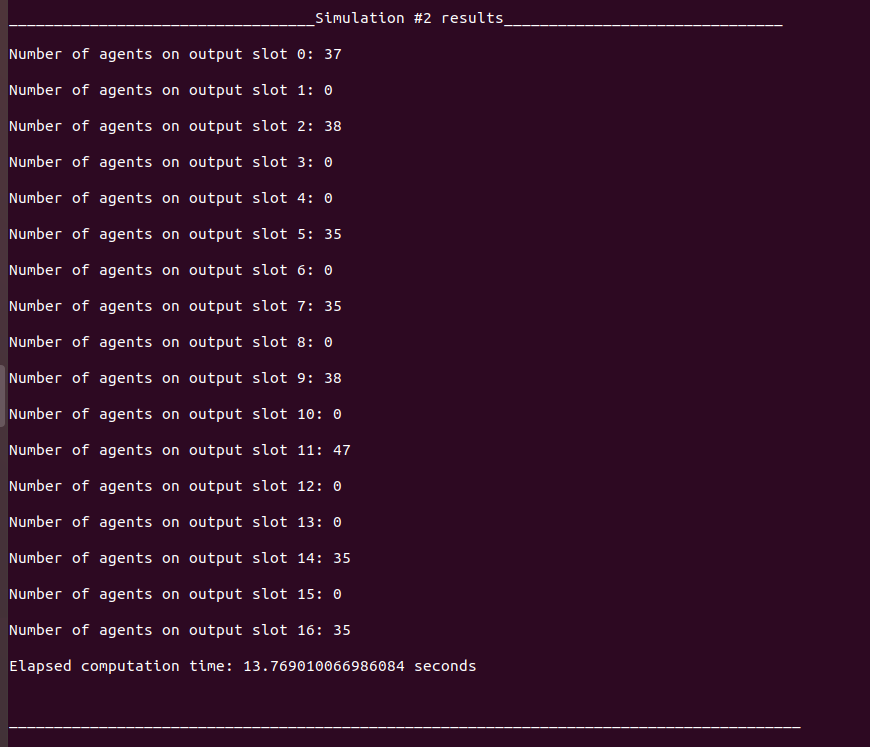
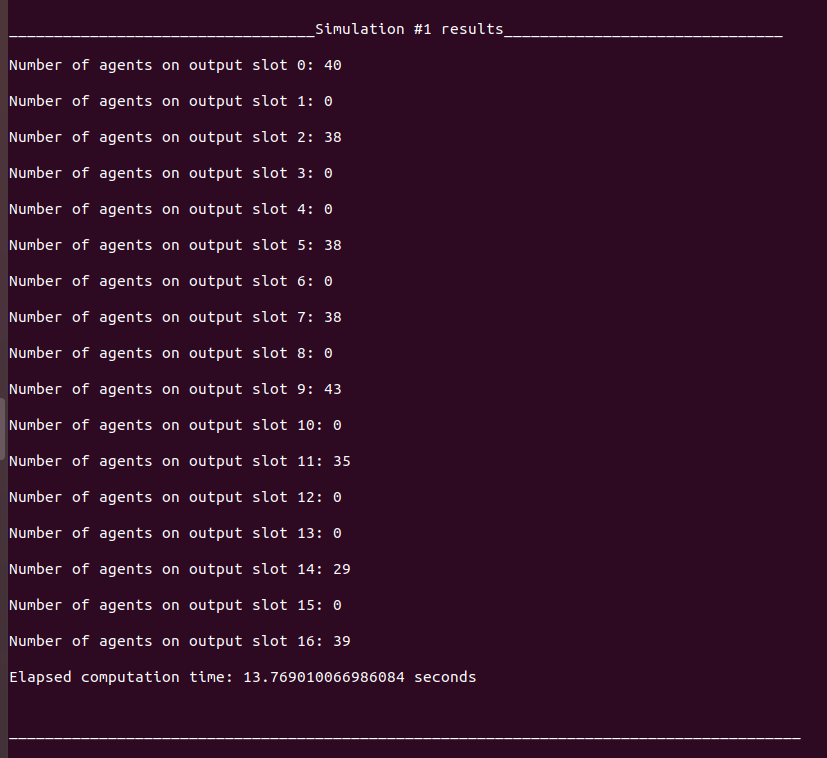
Manual SSP : manually enter the input numbers of the SSP problem

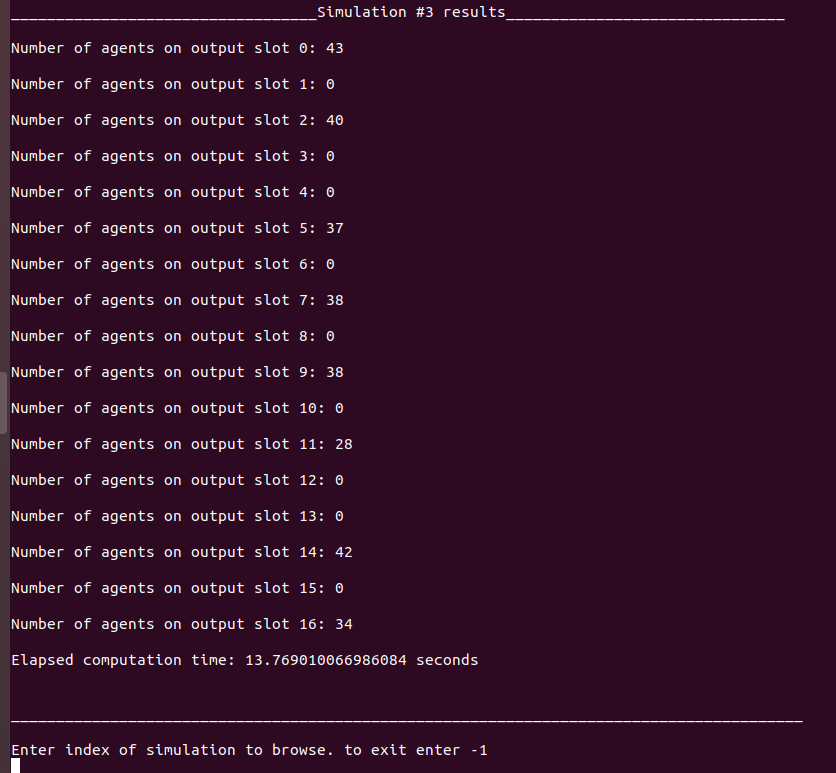
**Manual SSP:**

This mode allows the user to simulate a network that solves the SSP.

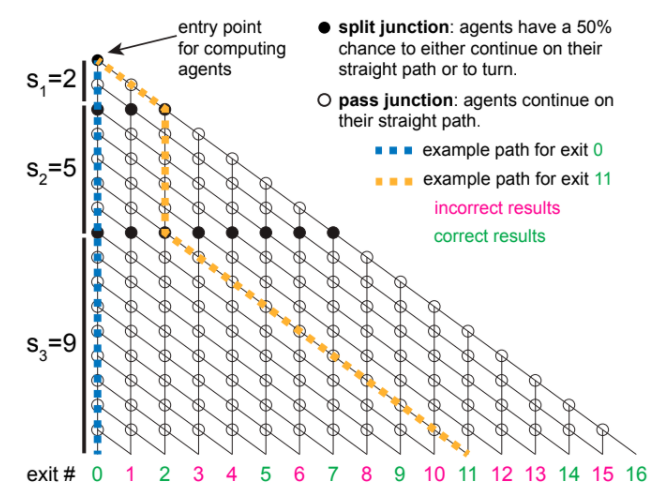
We used BCS to synthesize the computational network and simulate the runs of agents through it.







In this example we simulated the following network:



We chose a set of length 3 which contains the numbers {2, 5, 9}.

The amount of simulated agents was 300, and we chose to tag them.

In this case we used one thread for the calculation, and performed three different simulations (with the same parameters).

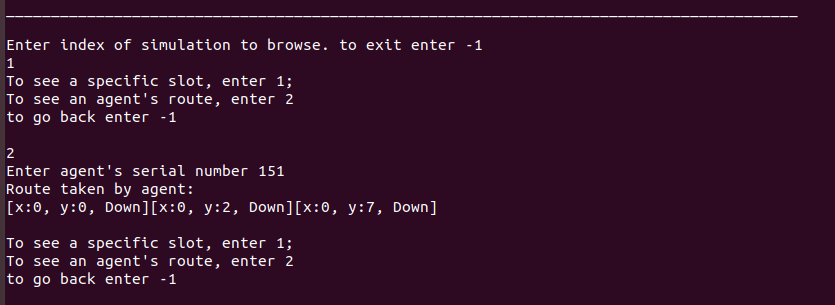
When the simulation is relatively big, increasing the number of threads may improve the run-time of the simulation. As written in Mboemo's BCS manual:

"Simulations can be run independently on separate threads, so multithreading can speed up runtimes considerably. We recommend using as many threads as you have available if the simulation is large."

Source: <https://beacon-calculus-simulator.readthedocs.io/en/latest/simulation.html>

After the simulation has ended, we can browse through the results, see how many agents ended up in each output slot, and watch each agent's full route in the network.

For example, to watch the route that agent number 151 in simulation #1 took:

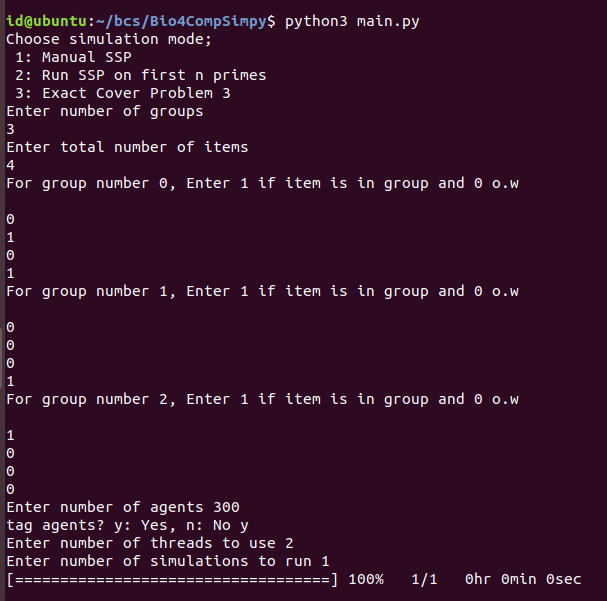


In this case we see that the agent took only down turns, meaning he chose the empty set {} so he ended up in slot 0.

**SSP on first N primes:**

This is pretty straight-forward, instead of manually choosing the numbers for the SSP, the user inputs on how many first prime numbers he wants to run the simulation on.

**Exact Cover:**

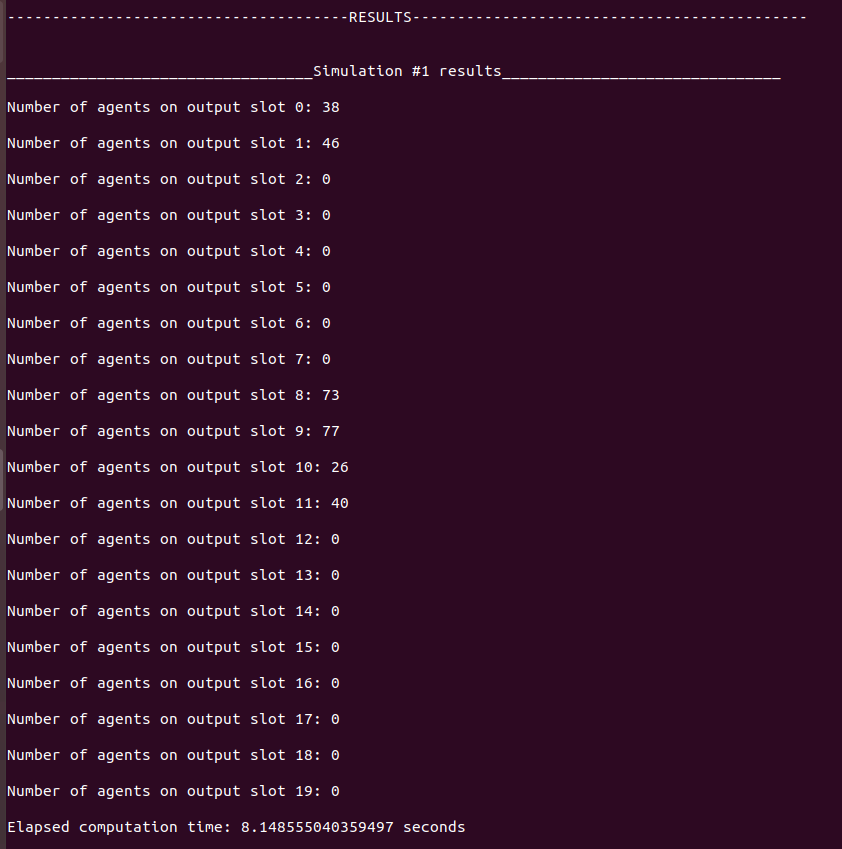


In this example we simulated a network solving the ECP with the following sets:

[1010] (=10), [1000](=8), [0001](=8) (The first digit in each group input is the LSB)

These are 3 groups with 4 possible items. Choosing the same item twice is illegal, so the network blocks some of the junctions thereby forcing agents in these junctions to take the down split, which means not choosing that specific group.

The simulation included 300 agents with tags, and was run using 2 threads.



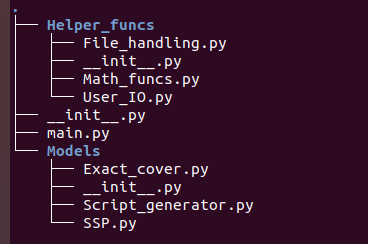
As in the other simulation options we can browse through the results and the agents routes.

**Plotting the results**

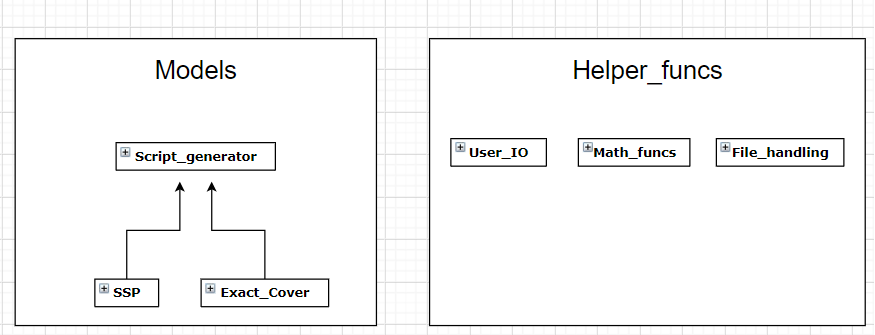
For each simulation, the program creates a .png file of the results graph (i.e number of agents in each slot)

**Programmers manual - Code Overview:**

Structure of the working directory:



Packages and Classes relations:



Models

The script\_generator class has all the basic fields and methods that are needed for any Bio4Comp device Beacon Calculus simulation.

If one wishes to create a bcs for some new computational network, he should define a new class that inherits from Script\_generator class, and override its "create\_bcs\_code()" function to set the bc code that synthesizes the network he wants.

As for now, we believe the most right and general BC code that simulates a computational geometrical network has the following structure:

* **Constants declaration:**
* Define number of agents, bumping rates, etc.
* **Processes definition:**
* "Setup process": this BC process synthesizes the computational network by using beacons to simulate all of the different junctions. It does so by a series of "beacon activation" actions, followed by an action that activates a beacon on a channel named "start" that signals to the agents processes the network is ready and they can start running.
* "Agent process": this BC process simulates the behavior of an agent exploring the network. Here the programmer needs to define the behavior of the agents as they move inside the net, such as what turns to take at junctions and at what rate, bumping with other agents (by using handshake actions) etc.
* **Starting conditions:**
* Set parameters for the setup process
* Set agent at network entries

**In code:**

bcs\_code =

self.print\_constants() + '''

Setup[''' + self.print\_setup\_params() + '''] = \n \t \t \t \t''' + self.print\_setup() + '''.{start![1] ,fast};

P[x,y,sum,last,serial] = {start?[1], r}.

(#DEFINE AGENT BEHAVIOR#);

Setup[''' + self.print\_partial\_sums() + ''']''' + "\n" + self.print\_processes\_start() + ";"

self.bcs\_code = bcs\_code

**SSP BC Model explanation:**

At first, the setup process is started. It produces beacons at y levels that

correspond to the partial sums of the numbers given, these are the split junctions.

These beacons transmit the value 0.

After the setup is done, the agents are created, each with a unique serial number (if the "tagged" option is on) , and then begin to travel in the network.

At each step an agent checks his y level:

If there is no active beacon on this y channel, then the agent proceeds to travel in the last direction that he took.

If there is an active beacon on this y channel, then it means the agent reached a split junction,

and then there is a 50% chance that he will split down, and 50% chance that he will split diagonally. The direction chosen is saved in the agent's "last" parameter

When an agent reaches a y level that is equal to the SSP numbers sum, it creates a beacon

on a channel called "done" and transmits through it his x level.

**Agent behavior in code:**

P[x,y,sum,last,serial] = {start?[1], r}.

(

[y==sum]->{done![x],fast} ||

[y!=sum]->{y?[0..sum], fast}.({splitDown,1}.P[x,y+1,sum,0,serial] + {splitDiag,1}.P[x+1,y+1,sum,1,serial]) ||

[y!=sum]->{~y?[0..sum],fast}.( [last == 0] -> {continueStraight, r}.P[x,y+1,sum,0,serial] +

[last == 1]-> {continueStraight, r}.P[x+1,y+1,sum,1,serial])

);

**Exact cover BC Model explanation:**

At first, the setup process is started. It produces beacons at y levels that

correspond to the partial sums of the sets encoding. For each set there is a calculation

of what x level bits don't override the set encoding. A set transmits over it's beacon

an x value if and only if this x value bits do not override the set encoding's bits.

After the setup is done, the agents are created, each with a unique serial number, and then

begin to travel the network. At each step an agent checks his y level:

1. If there are no active beacons on this y channel that transmit his x value, then the agent proceeds to travel in the last direction that he took.

2. If there is an active beacon on this y channel that transmits his x value, then it means the agent reached a split junction, and then there is a 50% chance that he will split down, and 50% chance that he will split diagonally.

3. If there is an active beacon on this y channel but it does not transmit x, then it means the agent has reached a blocked junction, then he will take the down split.

When an agent reaches a y level that is equal to the exact cover groups encoding sum, it creates a beacon on a channel called "done" and transmits through it his x level.

**Agent behavior in code:**

P[x,y,sum,last,serial] = {start?[1], r}.

(

[y==sum]->{done![x],fast} ||

[y!=sum]->{y?[x], fast}.({splitDown,fast}.P[x,y+1,sum,0,serial] + {splitDiag,fast}.P[x+1,y+1,sum,1,serial]) ||

[y!=sum]->{~y?[0],fast}.( [last == 0] -> {continueStraight, fast}.P[x,y+1,sum,0,serial] +

[last == 1]-> {continueStraight, fast}.P[x+1,y+1,sum,1,serial]) ||

[y!=sum]->{y?[0],fast}.{~y?[x],fast}.{blockedSplitDown,fast}.P[x,y+1,sum,0,serial]

);