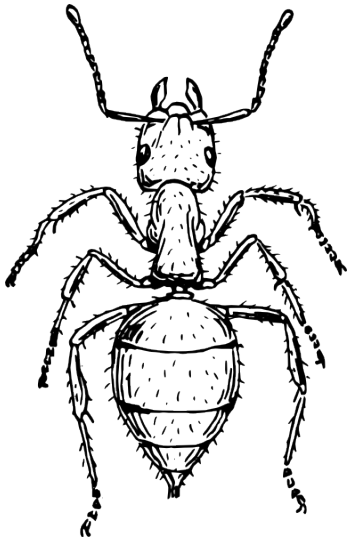
MODELO CONCURRENTE PARA SIMULAR EL COMPORTAMIENTO DE UNA COLONIA DE HORMIGAS EN *ERLANG/OTP*



JUNIO, 2018

DAVID LILUE

UNIVERSIDAD POLITÉCNICA DE MADRID

MÁSTER EN SOFTWARE Y SISTEMAS

Índice

1 INTRODUCCIÓN

Este trabajo tiene como objetivo simular el comportamiento de la hormigas al momento de buscar los caminos más cortos desde un punto A a otro B. La idea esta fundamentada en el algoritmo de optimización por colonia de hormigas[1] y no se tiene como objetivo implementar dicho algoritmo. El enfoque que se desea dar tiende a implementar y observar el comportamiento de distintos actores en un modelo concurrente. Usando pequeños agente que no tienen indicios inteligencia, a estos se les asigna una misión y simplemente ellos harán lo posible por lograrlo. A través del paso de mensajes entre distintos procesos, se plantea un modelo para simular el comportamiento de las hormigas en un grafo.

Enunciando el problema que se desea abarcar, y de donde ha surgido la naturaleza del mismo. Detallando el enfoque que este trabajo quiere presentar, además del uso de herramientas acordes al problema. Posteriormente, se describe el modelo propuesto para implementar el sistema concurrente, así como los distintos actores y sus responsabilidades. Finalizando con una breve conclusión acerca de distinta pruebas realizadas y motivos de los resultados obtenidos.

2 MOTIVACIÓN Y EL PROBLEMA

Comenzando con una breve descripción del problema que este trabajo tiene intenciones de abordar, además de las motivaciones que impulsan el análisis y desarrollo del modelo. El algoritmo de la colonia de hormigas es una técnica para encontrar caminos buenos en un grafo a través de métodos probabilísticos. Perteneciente a los métodos de inteligencia de enjambre, fue propuesto por Marco Dorigo en 1992[1] con el objetivo principal de encontrar el camino óptimo de un grafo.

La naturaleza de las hormigas puede describirse como sistema emergente, concepto filosófico que hace referencia a la auto-organización y supervivencia de actores, que por si solos no presentan aptitudes de inteligencia o que puedan desempeñar una labor en solitario. A pesar de eso, en conjunto logran convertirse en una inteligencia colectivo y desempeñar actividades extraordinarias.

Tomando en consideración estos conceptos, proponer un sistema concurrente suena tentador y así se puede visualizar el comportamiento de actores dependiendo del modelo que se proponga. La idea de este trabajo es ser exploratorio y probar conceptos a través de herramientas idóneas que permitan implementarlos. Progresivamente se presentarán problemas, decisiones y resultados que surgieron a lo largo del trabajo.

El problema que se propone resolver es *TSP*, o mejor dicho, una versión modificada de este, donde se busca el mejor camino para ir de un nodo a otro pasando por todos los nodos. Estos nodos se indican

directamente a las hormigas como una misión de buscar comida y regresar.

3 ERLANG/OTP

Toda el desarrollo del modelo propuesto ha sido implementado en `Erlang`, un sistema que permite construir soluciones para problemas que involucren concurrencia y comunicación entre distintos actores. Por eso y muchas razones es la herramienta por excelencia para abordar este tipo de problemas y en concreto al que se presenta en este trabajo.

En las próximas secciones se describirá el desarrollo de las distintas funciones y procesos que logran modelar un sistema de hormigas, así como el enfoque que se dio a la representación del problema e inconvenientes que se presentaron.

4 MODELO CONCURRENTE DE HORMIGAS

Para modelar el comportamiento de la colonia de hormigas se pensó inicialmente en cada hormiga como proceso fundamental en la implementación de la solución. Además de eso, debía existir un agente que mantuviera la información del grafo, implementado como un proceso maestro que tuviese el conocimiento, comunicación y poder sobre las hormigas.

En este punto surgió el primer inconveniente, la representación del grafo podía ser de forma convencional a través de una matriz o lista de adyacencia, o implementar un proceso para cada nodo; inclusive los arcos. A estos se les delegaría la obligación de comunicarse con las hormigas mientras estas se movían por el grafo, además de mantener la información de los rastros de feromonas y su evaporación.

Como un primer acercamiento, se decidió usar la versión clásica de grafos, esto permitía tener acceso directo desde un proceso a la información del estado actual de las hormigas y feromonas. A pesar de ello, delegar tantas responsabilidades a un solo proceso parecía ir en contra de lo que busca el paradigma concurrente. Por eso, el sistema termina implementando un proceso que se encarga de los nodos y sus vecinos, creando uno por cada nodo tenga el grafo. Las conexiones y pesos de los arcos terminan siendo una lista de 2-tuplas con el `PID` del vecino y el costo de ir hasta él.

El nodo maestro, mantiene una lista de los nodos y otra de las hormigas pero no interactúa con ellos. Solo los consulta, comienza y termina con la ejecución. La hormiga tiene una función esencial, cuando comienza a moverse, entra en un bucle que pregunta a su nodo anfitrión el siguiente paso y así sucesivamente. El la siguiente sección se puede ver con más detalle la implementación de cada agente en el modelo.

5 ACTORES DEL SISTEMA

Como se ha dicho previamente, este sistema concurrente está compuesto por tres tipos de actores fundamentales. Las hormigas, los nodos y un maestro. La interacción se concentra entre las hormigas y los nodos, entre hormigas no existe una comunicación más allá del rastro de feromonas. Concepto inspirado por su comportamiento natural. A continuación se puede ver parte de la implementación de como se ha definido a la hormiga.

```
ant(Node, Visited, Ns, Target) ->
    receive
        {init} ->
            Node ! {ask, self()},
            ant(Node, Visited, Ns, Target);
                                ⋮
```

En esta primera parte, se puede ver que un hormiga está constituida por un nodo anfitrión, los nodos visitados, la cantidad de nodos en el grafo y un nodo objetivo. Para que la hormiga comience a recorrer el grafo, es necesario mandar un mensaje al nodo anfitrión preguntando por los posibles caminos que puede escoger la hormiga. La decisión del camino es aleatoria aunque también dependerá de la cantidad de feromonas que exista en él.

Una decisión importante ha sido definir el comportamiento de la hormiga durante su recorrido en el grafo, hay casos donde puede encontrarse con el nodo objetivo sin haber visitado todos los nodos, existe la posibilidad de encontrarse atrapada porque no los únicos caminos disponibles conducen a nodos que ya han sido visitados y esto conlleva a una tener que tomar una decisión que logre modelar

la realidad. El modelo tiene como objetivo ser lo más realista posible, por ello en una situación como la anterior, la hormiga se considera perdida y en la naturaleza eso implica seguramente su fallecimiento porque su vida a perdido propósito y rumbo.

En el segmento de código que se muestra a continuación se trata de conseguir el comportamiento descrito el párrafo anterior.

```
{goto, NBH} when is_list(NBH) ->
    CurrentVisited = [Node|Visited],
    Banned = [Target|CurrentVisited],
    Edges = posibleEdges(NBH, Banned),
    if
        Edges == [] ->
            if
                length(Banned) == Ns ->
                    {_, Weight} = chooseOneOf(
                        posibleEdges(NBH, CurrentVisited)),
                    walk(self(), Node, Target, Weight),
                    % Go back with food
                    ant(Target, [], Ns, lists:last(CurrentVisited));
                true ->
                    [Previous|_] = Visited,
                    Previous ! {evaporate, Node},
                    Node ! {evaporate, Previous},
                    self() ! {init},
                    % Respawn back to source (ant died)
                    ant(lists:last(CurrentVisited), [], Ns, Target)
            end;
        true ->
            {NewNode, Weight} = chooseOneOf(Edges),
            walk(self(), Node, NewNode, Weight),
            ant(NewNode, CurrentVisited, Ns, Target)
    end;
```

NODOS Estos tienen la labor de indicar los posibles caminos que una hormiga puede usar y mantener un acumulador de tráfico. Una forma de saber la frecuencia con la que se visita ese nodo. Además de eso, mantienen un factor de feromonas para cada arco que posean. En este aspecto yace una decisión importante, el nivel de feromonas puede pertenecer al arco y al momento que una hormiga escoja ir por él, se incrementa dicho factor. Eso implica que, a pesar de la espera que debo realizar la hormiga, el arco ya tendría un nuevo valor y el caso inverso conlleva a que las feronomas se actualicen tarde. Lo que podría ocasionar alteración en las decisiones de otras hormigas.

Por esto, los arcos están divididos en dos, una sección para cada nodo perteneciente al mismo. La actualización de las feromonas ocurre en el nodo de partida cuando se escoge un camino y lo mismo ocurre en el nodo de llegada pero al momento que la hormiga llega a este. En la sección de código anterior podemos ver una llamada a una función llamada `walk`, esta está definida en el módulo de los nodos, se puede ver a continuación y es donde se resuelve el problema descrito con anterioridad.

```
walk(Walker, From, To, Weight) ->
    spawn(
        fun() ->
            From ! {ant_choose, To, Weight},
            timer:sleep(Weight * ?DELAY_FACTOR),
            timer:sleep(Weight * ?DELAY_FACTOR),
            To ! {ant_choose, From, Weight},
            To ! {ask, Walker}
        end
    ).
```

Para entender un poco más acerca del comportamiento de los nodos, su definición se puede ver a continuación. Un linea importante es donde se llama a la función `evaporatePheromone`, esta evaporación de las feromonas ocurre después de cierto tiempo definido y es lo que brinda más realismo al modelo.

```
node(NBH, Traffic) ->
    receive
        {add, NBR, Weight} ->
            node([{NBR, Weight, 1}|NBH], Traffic);
        {ask, Ant} ->
            Ant ! {goto, NBH},
            node(NBH, Traffic + 1);
        {ant_choose, Node, Weight} ->
            NewNBH = strengthenPheromone(Node, NBH, Weight),
            evaporatePheromone(self(), Node, Weight),
            node(NewNBH, Traffic);
        {evaporate, Node, Weight} ->
            NewNBH = weakenPheromone(Node, NBH, Weight),
            node(NewNBH, Traffic);
        _ -> node(NBH, Traffic)
    end.
```

Aquí se puede ver que existe una comunicación recursiva con las hormigas, cuando una hormiga manda un mensaje con la tupla `{ask, Ant}`, el nodo envía un mensaje donde la hormiga volverá a mandar mensaje `ask` pero a otro nodo. Este es el punto fundamental en la interacción de los actores en el modelo propuesto.

MAESTRO Su función es bastante sencilla, dado que varias de las responsabilidades que se le habían asignado fueron delegadas a los nodos. En resumen, tiene 3 atributos: número de nodos, una lista de

los `PID` correspondiente a cada nodo del grafo y una lista de hormigas. Además, reconoce tres mensajes relevantes: crear hormigas, iniciar recorrido de las hormigas y exterminar a las hormigas; junto a otras funciones auxiliares.

```
master(N, NodeList) -> master(N, NodeList, []).
master(N, NodeList, Ants) ->
    receive
        {init} ->
            awakening(Ants),
            master(N, NodeList, Ants);
        {createAnts, N_ants, Source, Target} ->
            exterminate(Ants),
            NewAnts = wakeUp(N_ants, Source, N, Target, []),
            master(N, NodeList, NewAnts);
        {killAnts} ->
            exterminate(Ants),
            master(N, NodeList, []);
        {printNodes} ->
            io:format("Graph: ", []),
            lists:map(fun (Nd) -> Nd ! {print} end, NodeList),
            master(N, NodeList, Ants);
        {printAnts} ->
            io:format("Ants: ", []),
            lists:map(fun (A) -> A ! {print} end, Ants),
            master(N, NodeList, Ants);
        _ ->
            master(N, NodeList, Ants)
    end.
```

Es evidente que se puede omitir este actor del modelo pero resulta cómodo para manejar los elementos del sistema. Como puede verse en la sección de código anterior, no existe una interacción real entre

este actor y otro en el modelo. Cabe destacar que la creación del grafo, y de los procesos, se lleva a cabo por funciones auxiliares que leen los datos desde un archivo y se crea la representación abstracta siguiendo el modelo descrito.

6 RESULTADOS Y CONCLUSIONES

Después de haber implementado el modelo concurrente, buscando mantener un sentido realista del comportamiento de las hormigas y fundamentar el desarrollo en el algoritmo *per se*. Se realizaron distintas pruebas, desde las más sencillas hasta otras con mayor dimensión, en el caso básico con dos nodos, y dos caminos de pesos diferentes, se pudo ver que el camino más corto tener mayor flujo de hormigas y un nivel de feromonas elevado. A pesar de eso, el camino largo seguía teniendo afluencia de hormigas y esto nos puede decir que el comportamiento resulta como es esperado.

En casos más grande se puede ver que la afluencia de hormigas es más dispersa pero de igual forma nos brinda información de los arcos que pueden descartarse porque el nivel de feromonas es muy reducido. En ciertas ocasiones se puede ver como dos arcos que parten del mismo nodo tienen una diferencia considerable de feromonas y sus estados se invierten. Esto puede suceder porque los pesos de los arcos es equilibrado o por simple aleatoriedad del sistema.

El algoritmo que presenta este trabajo tiene desventaja sobre la versión original, y es el hecho de desconocer el peso del camino antes decidir cual cursar. Dejar a las hormigas ciegas de esta información

se debe a optar por una simulación más realista, y a pesar de esto, sigue dando respuestas acorde al objetivo inicial; encontrar un buen camino.

REFERENCIAS

[1] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed optimization by ant colonies, 1991.