UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Aprendizaje Automático, Computación Evolutiva e Inteligencia de Enjambre para Aplicaciones de Robótica

Trabajo de graduación presentado por Gabriela Iriarte Colmenares para optar al grado académico de Licenciado en Ingeniería Electrónica

Guatemala,

2020

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Aprendizaje Automático, Computación Evolutiva e Inteligencia de Enjambre para Aplicaciones de Robótica

Trabajo de graduación presentado por Gabriela Iriarte Colmenares para optar al grado académico de Licenciado en Ingeniería Electrónica

Guatemala,

2020

Vo.Bo.:

(f) _____

Dr. Luis Alberto Rivera Estrada

Tribunal Examinador:

(f) _____

Dr. Luis Alberto Rivera Estrada

(f) _____

MSc. Carlos Esquit

(f) _____

MSc. Miguel Enrique Zea Arenales

Fecha de aprobación: Guatemala, 5 de diciembre de 2020.

Prefacio

La idea aquí es más filosófico de qué es y por qué se está haciendo. Es menos técnico. Interés en el área. Agradecimientos y dedicatorias.

Índice

Lista de figuras

Lista de cuadros

Resumen

Aquí va todo.

Abstract

This is an abstract of the study developed under the

CAPÍTULO 1

Introducción

Implementar y verificar algoritmos de inteligencia de enjambre y computación evolutiva como alternativa al método de Particle Swarm Optimization para los Bitbots de UVG. Implementar el algoritmo Ant Colony Optimization (ACO) y encontrar el valor de los parámetros de las ecuaciones del ACO que permitan a los elementos de la colonia encontrar una meta específica, en un tiempo finito. Validar los parámetros encontrados por medio de simulaciones computarizadas que permitan la visualización del comportamiento de la colonia. Adaptar los modelos del movimiento y de la cinemática de los Bitbots, desarrollados en la fase anterior del proyecto, al algoritmo ACO. Validar el algoritmo ACO adaptado implementándolo en robots físicos simulados en el software WeBots, y comparar su desempeño con el del Modified Particle Swarm Optimization (MPSO).

CAPÍTULO 2

Antecedentes

2.1. Megaproyecto Robotat - Fase II

En la segunda fase del proyecto Robotat [1] se elaboró un algoritmo basado en el Particle Swarm Optimization (PSO) para que distintos agentes llegaran a una meta definida (el mínimo de una función de costo). El algoritmo modificado elaboraba una trayectoria a partir de la actualización de los puntos de referencia a seguir, generados por el PSO clásico. Asimismo, se realizó distintas pruebas para encontrar los mejores parámetros del algoritmo PSO, probando distintas funciones de costo. Para realizar dicho algoritmo se tomó en cuenta que la velocidad de los robots diferenciales tiene un límite, además de tener que seguir las restricciones físicas de un robot real. Para simular estas restricciones se utilizó el software de código abierto Webots luego de haberlo implementado en Matlab como partículas sin masa ni restricciones. Además, en [2] se realizó experimentos con el mismo algoritmo, pero utilizando Virtual Potential Fields como forma de evasión de obstáculos.

En Webots también se implementó distintos controladores para que la trayectoria fuera suave y lo más uniforme posible. Entre los controladores que se implementaron están: Transformación de uniciclo (TUC), Transformación de uniciclo con PID (TUCPID), Controlador simple de pose (SPC), Controlador de pose Lyapunov-estable (LSPC), Controlador de direccionamiento de lazo cerrado (CLSC), Transformación de uniciclo con LQR (TUC-LQR), y Transformación de uniciclo con LQI (TUC-LQI). El controlador con el mejor resultado en esta fase fue el TUC-LQI.

2.2. Robotarium de Georgia Tech

El Robotarium del Tecnológico de Georgia en Estados Unidos desarrolló una mesa de pruebas para robótica de enjambre para que personas de todo el mundo pudieran hacer pruebas con sus robots. De este modo, ellos están apoyando a que más personas, sin importar sus recursos económicos, puedan aportar a la investigación de robótica de enjambre. Además,

eso ayudaría a los investigadores a dejar de simular y probar sus algoritmos en prototipos reales. Para utilizar la plataforma hay que descargar el simulador del Robotarium de Matlab o Python, registrarse en la página del Robotarium y esperar a ser aprobado para crear el experimento [3].

2.3. Wyss Institute Swarm Robots

El Instituto Wyss de Harvard desarrolló robots de bajo costo miniatura llamados *Kilobots* para probar algoritmos de inteligencia computacional de enjambre. Este tipo de robots es desarrollado para potencialmente realizar tareas complejas en el ambiente como lo son polinizar o creación de construcciones. Estos Kilobots cuentan con sensores, micro actuadores y controladores robustos para permitir a los robots adaptarse a los ambientes dinámicos y cambiantes [4].

2.4. Aplicaciones de Ant Colony Optimization (ACO)

En [5] puede encontrarse un trabajo similar, donde se aplica el método de ACO en un robot autónomo en una cuadrícula con un obstáculo estático. El robot lo colocan en la esquina inferior izquierda y se pretende que alcance el objetivo en la esquina superior derecha de la cuadrícula. La simulación de este trabajó se realizó en Matlab y se implementó una cuadrícula de 100x100 con la unidad como el tamaño de las aristas. Además, en la cuadrícula los agentes podían moverse norte, sur, este, oeste y en forma diagonal.

También en [6] se realizó una comparación de los algoritmos PSO y ACO, en donde se resaltó algunas ventajas y desventajas de cada uno de los métodos. El PSO es simple pero sufre al quedarse atascado con mínimos locales por la regulación de velocidad. Por otro lado, el ACO se adapta muy bien en ambientes dinámicos, pero el tiempo de convergencia puede llegar a ser muy largo (aunque la convergencia sí está garantizada).

CAPÍTULO 3

Justificación

En la segunda fase del proyecto Robotat de la Universidad del Valle de Guatemala se desarrolló un algoritmo basado en el algoritmo Particle Swarm Optimization para planificar una trayectoria óptima para que los robots llegaran a una meta determinada. También se elaboró el diseño de distintos controladores para asegurar que los robots recibieran velocidades coherentes con respecto a sus limitantes físicas. Ya que este algoritmo y controlador fueron exitosos en simulación, se desearía comparar con otros algoritmos para que, de este modo, el mejor algoritmo sea finalmente implementado en los robots físicos.

Como propuesta de algoritmo se tiene el Ant Colony Optimization(ACO), pues es otro de los más utilizados de la rama de inteligencia computacional de enjambre. Se debe de encontrar los mejores parámetros para que el algoritmo converja en un tiempo similar al logrado con el PSO de la fase II del proyecto Robotat. Asimismo se busca implementar los mismos controladores que en el proyecto mencionado anteriormente para que la comparación sea equitativa.

CAPÍTULO 4

Objetivos

4.1. Objetivo General

Implementar y verificar algoritmos de inteligencia de enjambre y computación evolutiva como alternativa al método de Particle Swarm Optimization para los Bitbots de UVG.

4.2. Objetivos Específicos

- Implementar el algoritmo Ant Colony Optimization (ACO) y encontrar el valor de los parámetros de las ecuaciones del ACO que permitan a los elementos de la colonia encontrar una meta específica, en un tiempo finito.
- Validar los parámetros encontrados por medio de simulaciones computarizadas que permitan la visualización del comportamiento de la colonia.
- Adaptar los modelos del movimiento y de la cinemática de los Bitbots, desarrollados en la fase anterior del proyecto, al algoritmo ACO.
- Validar el algoritmo ACO adaptado implementándolo en robots físicos simulados en el software WeBots, y comparar su desempeño con el del Modified Particle Swarm Optimization (MPSO).

CAPÍTULO 5

Alcance

Podemos usar Latex para escribir de forma ordenada una fórmula matemática.

CAPÍTULO 6

Marco teórico

6.1. Inteligencia Computacional de Enjambre

El campo de la inteligencia computacional de enjambre, o *Swarm*, parte de la idea de que individuos interactúan entre ellos para resolver un objetivo global por medio de intercambio de de su información local. De esta manera, la información se propaga más rápido y por tanto, el problema se resuelve de una manera más eficiente. El término *Swarm Intelligence* (SI) se refiere a esta estrategia de resolución de problemas colectiva, mientras que *Computational Swarm Intelligence* (CSI) se refiere a algorimos que modelan este comportamiento [7].

Al tener diversos estudios sobre distintos tipos de animales es posible crear algoritmos que modelen las situaciones. Algunos sistemas biológicos que han inspirado este tipo de algoritmos son: hormigas, termitas, abejas, arañas, escuelas de peces y bandadas de pájaros. Dos de los algoritmos que se estudiarán más adelante están basados en pájaros y hormigas, modelando su comportamiento individual y colectivo.

6.2. Particle Swarm Optimization (PSO)

Este algoritmo hace referencia al comportamiento de las bandadas de pájaros y cardúmenes de peces al realizar búsquedas óptimas [8]. Las partículas encuentran la solución a partir de su mejor posición y la mejor posición de todas las partículas [9]. Las ecuaciones que modelan esta situación son las siguientes [8], [9]:

$$v_{id}^{t+1} = v_{id}^{t+1} + c_1 r_1 (p_{id}^t - x_{id}^t) + c_2 r_2 (p_{gd}^t - x_{id}^t) \quad (1)$$

$$x_{id}^{t+1} = x_{id}^t + v_{id}^{t+1} \quad (2)$$

Donde v es la velocidad, x la posición, c son constantes de aceleración, p_{id} es la mejor

posición personal, p_{gd} es la mejor posición que tuvo todo el enjambre en la iteración y r son números aleatorios entre 0 y 1. La fase de exploración ocurre cuando se tiene una diferencia grande entre la posición de la partícula y el particle best (pbest) o global best (gbest). La ventaja de este método contra los que necesitan usar el gradiente es que no se requiere que el problema sea diferenciable y puede optimizarse problemas con ruido o cambiantes. La función que se desea optimizar en este caso se llama función de costo [8].

6.2.1. Funcionamiento del Algoritmo PSO

Se inicia el programa creando las partículas, dándoles una posición y velocidad inicial. Luego se introducen esos valores a la función de costo, se actualizan los valores de pbest y gbest. Este proceso se repite hasta que se cumpla un número de iteraciones o se logre la convergencia del algoritmo. La convergencia se alcanza cuando todas las partículas son atraídas a la partícula con la mejor solución (gbest). Un factor que se debe tomar en cuenta es el de restringir los valores que pueden tomar las funciones para que el algoritmo no diverja [9].

6.2.2. Mejora del Algoritmo

El algoritmo puede mejorarse a través del incremento de partículas, introducción del peso de inercia w o la introducción de un factor de constricción K. El factor de inercia controla las fases de explotación y desarrollo del algoritmo. Se sugiere utilizar un valor mayor de inercia y decrementarlo hasta un valor menor para tener una mayor exploración al principio, pero al final mayor explotación [8].

$$v_{id}^{t+1} = wv_{id}^{t+1} + c_1 r_1 (p_{id}^t - x_{id}^t) + c_2 r_2 (p_{gd}^t - x_{id}^t) \tag{3}$$

$$v_{id}^{t+1} = K(wv_{id}^{t+1} + c_1 r_1 (p_{id}^t - x_{id}^t) + c_2 r_2 (p_{gd}^t - x_{id}^t)) \tag{4}$$

El otro caso es utilizar el factor de constricción K para mejorar la probabilidad de convergencia y disminuir la probabilidad de que las partículas se salgan del espacio de búsqueda [8].

6.3. Ant Colony Optimization (ACO)

A partir de las observaciones de los entomólogos, se determinó que las hormigas tienen la habilidad de encontrar el camino más corto entre una fuente de alimento y su hormiguero. Marco Dorigo desarrolló un algoritmo con base en el comportamiento de estos insectos, a partir del cual se han derivado muchas otras variantes. Sin embargo, la idea principal de utilizar a las hormigas como base del algoritmo puede verse como una instancia de la metaheurística de Ant Colony Optimization (ACO-MH). Algunas de las instancias más conocidas son: Ant System (AS), Ant Colony System (ACS), Max-Min Ant Aystem (MMAS), Ant-Q, Fast Ant System, Antabu, AS-rank y ANTS [7].

Cuando una hormiga encuentra una fuente de alimento, al regresar al nido esta deja un rastro de feromonas para indicarle a sus compañeras que si siguen ese camino, encontrarán comida. Mientras más hormigas escojan ese camino, más fermona habrá y asimismo, mayor probabilidad de que las demás hormigas elijan esa ruta. Esta forma de comunicación indirecta que tienen las hormigas es denominada *stigmergy*. «La feromona artificial imita las características de la feromona real, indicando la popularidad de la solución del problema de optimización que se está resolviendo. De hecho, la feromona artificial funciona como una memoria a largo plazo del proceso completo de la búsqueda»[7].

Al principio, las hormigas se comportan de una manera aleatoria. Deneubourg realizó un experimento en el que se colocó comida a cierta distancia del hormiguero, pero solo podían acceder a ella en dos caminos diferentes. En este experimento, los caminos eran del mismo tamaño, pero con el comportamiento aleatorio de las hormigas uno de ellos fue seleccionado. Este experimento es conocido como el "puente binario". Asimismo, en un experimento similar pero con uno de los caminos más corto que el otro, las hormigas al principio eligieron los dos de forma equitativa, pero conforme el tiempo pasó eligieron el más corto. Este efecto es denominado "largo diferencial"[7].

6.3.1. Simple Ant Colony Optimization (SACO)

Este algoritmo es el que se utilizó en la implementación del experimento del puente binario. Este considera el problema general de búsqueda del camino más corto entre un conjunto de nodos en un grafo $G = (V, E)$ donde la matriz V representa a todos los vértices o nodos del grafo y la matriz E a todas las aristas o *«edges»* del grafo [7].

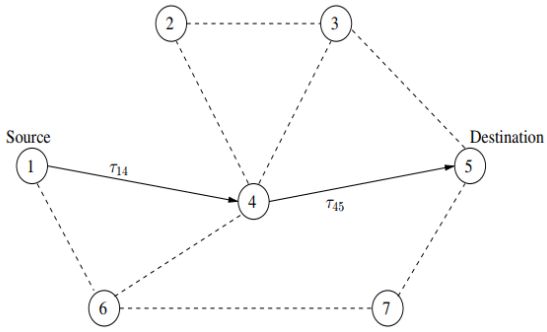


Figura 1: Ejemplo de un grafo [7].

Asimismo, el largo L^k del camino construido por la hormiga k se calcula como el número de saltos en el camino desde el nodo que representa al nido hasta el que representa el destino con la comida. En la figura 1 puede verse un camino marcado con flechas continuas con un largo de 2. En cada arista (i,j) del grafo se tiene cierta concentración de feromona asociada τ_{ij}. En este algoritmo, la concentración inicial de feromona se asigna de forma aleatoria[1] (aunque en la vida real sea de 0). Cada hormiga decide inicialmente de forma aleatoria[2] a qué arista dirigirse. K hormigas se colocan en el nodo fuente (source). Para cada iteración, cada

[1]Sin embargo en [10] recomiendan que todas las aristas comiencen con el mismo valor, como 1 por ejemplo.
[2]En [10] y [7] recomiendan usar el algoritmo Roulette Wheel.

hormiga construye una solución al nodo destino. En cada nodo i, cada hormiga k determina a qué nodo j debe de dirigirse basado en la probabilidad p:

$$p^k_{(i,j)}(t) = \begin{cases} \frac{(\tau_{ij}(t))^\alpha}{\sum\limits_{k \in J_k} (\tau_{ij}(t))^\alpha} & \text{si j} \in N^k_i \\ 0 & \text{si j} \notin N^k_i \end{cases} \tag{5}$$

Donde N representa al conjunto de nodos viables conectados al nodo i. Si en cualquier nodo el conjunto N es el conjunto vacío, entonces el nodo anterior se incluye. En la ecuación 5, α es una constante positiva que se utiliza para modular la influencia de la concentración de feromona. Cuando este parámetro es muy grande la convergencia puede ser extremadamente rápida y resultar en un camino no óptimo. Cuando las hormigas llegan al nodo destino regresan a su nodo fuente por el mismo camino por el cual llegaron y depositan feromona en cada arista [7]. En este proceso también se modela una forma de evaporación de feromona para aumentar la probabilidad de exploración del terreno.

La ecuación que gobierna la evaporación de la feromona en los trayectos es la siguiente:

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} (\Delta\tau^k_{ij}(t)) \tag{6}$$

Donde m es el número de hormigas, ρ es el factor de evaporación de feromona (entre 0 y 1). Además, se sabe que:

$$\Delta\tau^k_{ij}(t) = \frac{Q}{L_k}(t) \tag{7}$$

Donde Q es una constante y L es el costo del trayecto, como el largo del mismo. El valor representa el cambio de feromona entre el nodo i y j que la hormiga visitó en la iteración t [8].

6.3.2. Ant System

Este fue el primer algoritmo desarrollado que emula el comportamiento de las hormigas, que consiste en hacer ciertas mejoras al algoritmo presentado anteriormente. El algoritmo anterior tenía un objetivo más instructivo, por lo que era más simple. Algunas de las mejoras son la incorporación de información heurística en la ecuación de probabilidad y agregando capacidad de memoria con una lista tabú. La nueva ecuación de probabilidad es la siguiente:

$$p^k_{(i,j)}(t) = \begin{cases} p^k_{(i,j)}(t) = \frac{(\tau_{ij}(t))^\alpha (\eta_{ij}(t))^\beta}{\sum\limits_{k \in J_k} (\tau_{ij}(t))^\alpha (\eta_{ij}(t))^\beta} & \text{si j} \in N^k_i \\ 0 & \text{si j} \notin N^k_i \end{cases} \tag{8}$$

Donde el nuevo parámetro η representa el inverso del costo de la arista. En ambos algoritmos presentados, la condición de paro puede ser un límite de iteraciones o generaciones, o bien, cuando un porcentaje determinado de hormigas haya elegido la misma solución [7].

6.4. Computación Evolutiva

La evolución (en biología) puede ser definida como un proceso de optimización donde los individuos cambian de forma dinámica para sobrevivir en ambientes competitivos. Aunque Darwin se considera el padre de la la teoría de la evolución, fue Jean-Baptiste Lamark quien en realidad lo teorizó. Esta teoría consistía en que los padres heredan ciertas características a sus hijos y luego estos continúan adaptándose. Además, la teoría de selección natural de Darwin resume el hecho de que los individuos con mejores características tienen mayor probabilidad de supervivencia y de reproducirse. Por tanto, las características menos útiles y los individuos más débiles suelen desaparecer. La computación evolutiva (EC por sus siglas en inglés) se encarga de resolver problemas utilizando modelos de procesos evolutivos como la selección natural y supervivencia del más apto [7]. Dependiendo de la implementación, estos algoritmos pueden dividirse en los siguientes paradigmas:

- Algoritmos genéticos (GA)
- Programación genética (GP)
- Programación evolutiva (EP)
- Estrategias evolutivas (ES)
- Evolución diferencial (DE)
- Evolución cultural (CE)
- Co-evolution (CoE)

6.4.1. Algoritmos Genéticos (GA)

Los algoritmos genéticos son un tipo de algoritmo que utiliza de forma inteligente los conocimientos sobre evolución en biología para resolver problemas en distintos ámbitos de forma computacional. En esta evolución, las características de los individuos se expresan en genotipos y se tiene un operador de selección y otro de cruza o recombinación. El operador de selección se encarga de modelar la supervivencia de los individuos más aptos, mientras que el operador de recombinación modela la reproducción entre los individuos. Asimismo, la etapa de la recombinación o cruza se encarga de explotar las soluciones, mientras que la etapa de mutación se encarga de proveerle cierto grado de diversidad a la población para así evitar convergencia prematura a soluciones sub-óptimas [7].

Inicialmente se tiene una población de soluciones conformada por cromosomas (cada cromosoma representa una solución), los cuales están conformados por genes o características. Los valores que pueden tomar estos genes son los alelos y dependen de la codificación que se

le de al problema. ¿Por qué es necesario codificar? Pues principalmente porque necesitamos traducir las variables físicas con las que trabaja el problema para que la computadora pueda interpretarlas y así aplicarle los operadores genéticos [7].

6.4.2. Métodos de codificación

John Holland es considerado el padre de los algoritmos genéticos, luego de realizar trabajos con los mismos a partir de los trabajos previamente realizados por Bremermann y Reed. En este primer algoritmo genético canónico se utilizó una codificación binaria con formato de string, selección proporcional para elegir qué padres serán recombinados, cruza en un punto para producir "cromosomas hijos" y mutación uniforme [7].

Codificación binaria

Algunas ventajas de este tipo de codificación son que casi todos los tipos de hipótesis pueden ser planteadas utilizando números binarios y que las computadoras tienen como lenguaje primario al código binario. Por lo tanto, las computadoras realizan todas sus operaciones en binario por ser el único lenguaje que conocen y si le damos directamente números en su idioma el tiempo de procesamiento será más rápido. Para realizar el diseño de la codificación de un espacio de hipótesis con código binario se representan ciertos atributos utilizando sub conjuntos del mismo string de números binarios. Estos sub conjuntos (o genes en nuestro caso) pueden ser alterados sin necesidad de alterar a sus sub conjuntos vecinos [11].

Codificación entera

Además de la codificación binaria también existen codificaciones en base 10 como lo son la codificación de enteros y la de los reales. La codificación de enteros son muy convenientes en problemas que requieran como respuesta un orden específico de objetos ordenados. Un ejemplo donde es fácil visualizar la utilidad de este tipo de codificación es en el problema del vendedor viajero o TSP por sus siglas en inglés. Este problema busca encontrar el orden en el que el viajero debe de visitar ciertas ciudades dadas, tomando en cuenta que se debe de optimizar la ruta según la distancia de ciudad en ciudad. La codificación binaria podría utilizarse aquí, pero es posible que se requiera algún algoritmo corrector extra luego de hacer las operaciones de cruza o mutación pues algunos resultados pueden llevar a nodos (o ciudades) no existentes en la precisión dada por la longitud del número binario [12].

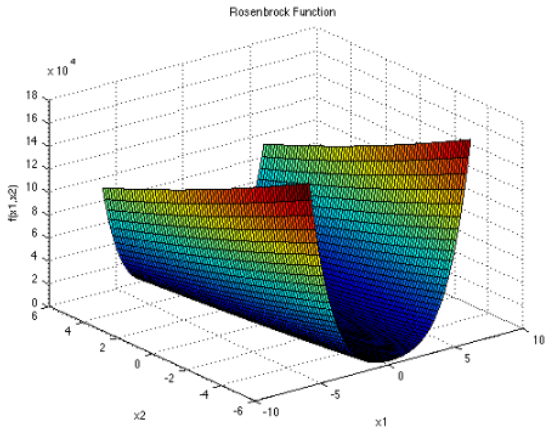
Codificación real

Finalmente tenemos la codificación de valores reales, que puede ser conveniente en problemas donde se esté trabajando en tiempo continuo, por lo que lo más intuitivo es utilizar números de punto flotante. Estos números claro que pueden ser expresados en binario, pero puede llegar a ser muy complicado su manejo. Por lo tanto, en estos casos donde la complejidad de la representación de los objetos es alta con números binarios se recomienda utilizar valores reales [13].

Como se mostró anteriormente, la codificación binaria es lo suficientemente robusta para representar todos los casos. Cabe recordar que nuestra computadora utiliza binario como lenguaje nativo y por lo tanto es posible codificar todo en este lenguaje. Sin embargo, en algunos casos es más intuitivo y fácil de calcular y manejar el problema utilizando números en base 10. Por lo tanto, se recomienda al lector apegarse a lo que le sea más fácil de interpretar, claro, sin comprometer la velocidad y complejidad del algoritmo.

6.4.3. Función de costo

Esta es la función que se desea minimizar (o maximizar) utilizando los algoritmos genéticos. Existen algunas funciones famosas utilizadas para probar algoritmos de optimización que tienen ciertas características como: muchos mínimos locales, forma de plato hondo, forma de valle, con gotas, etc. A continuación se presenta algunos ejemplos de este tipo de funciones [14].



$$f(\mathbf{x}) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

Figura 2: Gráfico y función de Rosenbrock [14].

6.4.4. Métodos de selección proporcional

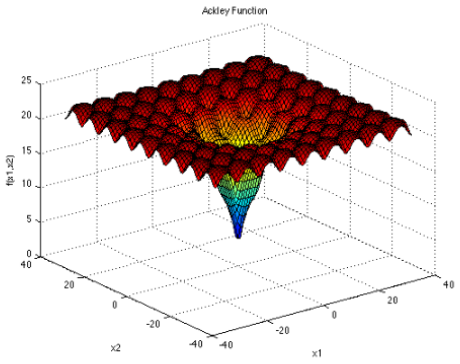
Determinístico

Estocástico

6.4.5. Métodos de selección tipo torneo

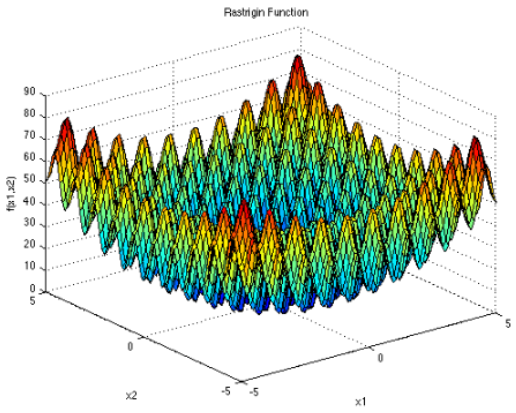
6.4.6. Operadores de cruza en GA

Existen tres clases principales de operadores de cruza o recombinación: asexual, sexual y multi-recombinación. En el caso se la asexual, los descendientes son generados a partir de



$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Figura 3: Gráfico y función de Ackley [14].

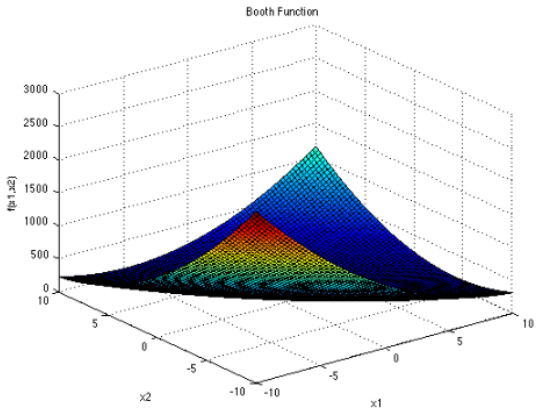


$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d}\left[x_i^2 - 10\cos(2\pi x_i)\right]$$

Figura 4: Gráfico y función de Rastrigin [14].

solamente un padre. En la segunda clase, los descendientes son generados a partir de dos padres seleccionados. Cabe mencionar que en este caso la cantidad de descendientes puede ser de 1 o 2. Finalmente, tenemos el caso de multi recombinación donde la única diferencia con el anterior es que puede producir más hijos (más de 2) si así se deseara. Aparte de esta clasificación de los operadores, también se tiene la clasificación por tipo de dato (binario, entero o decimal) que se utilizará en la codificación de los cromosomas [7].

En la mayor parte de los operadores binarios se utiliza la reproducción sexual con dos padres, donde debe especificarse qué bits serán intercambiados para crear a los dos hijos. Como se mencionó anteriormente, para codificar los cromosomas se utiliza el alfabeto finito binario, constituido por los valores de 0 y 1 únicamente. Este fue el método usado por Holland y el más simple de utilizar con strings de bits de un largo fijo. Algunas desventajas son el hecho de que el largo tiene que ser fijo y que por lo tanto, la precisión de la respuesta también está sujeta a dicho parámetro [11].



$$f(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$

Figura 5: Gráfico y función de Booth [14].

6.4.7. Métodos de mutación

[15][16][17][18]

6.5. Grafos

Como pudo observarse anteriormente, el algoritmo Ant System es un algoritmo de búsqueda basado en grafos y no funciones como el PSO. Por lo tanto, es necesario definir qué es un grafo y de qué maneras puede representarse. Un grafo G es un par de conjuntos (V, E) donde V representa al conjunto de vértices del grafo y E al conjunto de pares no ordenados de elementos de V [19].

$$V = \{V_1, V_2, \dots V_n\}$$
$$E = \{(V_i, V_j), (V_i, V_j), \dots\}$$

Existen los grafos dirigidos (o digrafos), donde el enlace del nodo i al j no es el mismo que del nodo j al i. Estos grafos sí cuentan con dirección. Asimismo, existen los grafos (y digrafos) ponderados. A estos se les asocia un número de peso o costo a cada arista. Este número podría ser distancia, capacidad o valor temporal dependiendo de la aplicación [20].

Los grafos pueden representarse a partir de dibujos como el de la figura 1, donde los grafos se muestran como en la figura y los digrafos con flechas para indicar la dirección. Además, también pueden representarse a partir de matrices como la matriz de adyacencia, En esta matriz cuadrada de tamaño número de vértices se muestra qué vértices están y no están conectados con un 1 y un 0 respectivamente [21]. A continuación se muestra un ejemplo de matriz de adyacencia de la figura 1.

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Como puede observarse, cada fila y columna representa a un vértice, empezando desde el vértice 1 en la primera fila y primera columna (esquina superior izquierda).

6.5.1. Grafos en Matlab

Para representar grafos en Matlab se utiliza la clase *graph*, ya disponible para utilizarse en Matlab. Estos objetos de la clase graph tienen métodos para acceder y modificar nodos y aristas, para mostrar el grafo en representación de matrices, mostrar información del grafo, visualización e incluso para encontrar el camino más corto entre nodos. Algunos de los atributos (o propiedades como les llama Matlab) son las aristas y los nodos, aunque es posible agregarle más atributos a la clase. Para mas información y documentación acerca de la clase *graph* de Matlab visitar [22].

6.6. Programación Orientada a Objetos

Como se explicó anteriormente, el software Matlab representa los grafos como objetos, por lo que es necesario saber lo básico de cómo funcionan estos para implementarlos. Primero, ¿Qué es un objeto? Un objeto es una colección de datos con ciertos comportamientos asociados. Un objeto puede ser de una clase en específico, que se define como una plantilla para crear objetos. El objeto en sí de una clase se llama instancia, y es el que se utiliza activamente en la programación. Para diferenciar objetos se utilizan atributos o características del mismo. Por ejemplo, un objeto de la clase fruta puede ser manzana o pera y se diferencian por sus atributos color y forma (por ejemplo) [23].

Como se explicó en el párrafo anterior, un objeto tiene comportamientos asociados, lo que nos hace pensar en ciertas funciones o acciones que pueden hacerse con el objeto. Estas funciones se llaman métodos y básicamente son funciones que solo pueden ser utilizadas por instancias de la clase donde están definidas [23]. Generalmente para referirse a un atributo se utiliza la notación «instancia.atributo» y para usar un método se utiliza «instancia.función()». Para conocer y aprender sobre la notación específica de programación orientada a objetos en Matlab puede verse [24].

6.7. Computación Paralela

Para realizar pruebas debe de correrse el programa varias veces, por lo que se consumiría demasiado tiempo dependiendo de la cantidad y forma de realizar las pruebas. Por lo tanto se propone utilizar programación paralela para aprovechar los distintos cores de las computadoras para acelerar el proceso [25].

Los softwares tradicionales se escriben para computación serial y no paralela, por lo que el problema se divide en pedazos pequeños y se van pasando uno por uno al procesador como puede verse en la figura 6 [25].

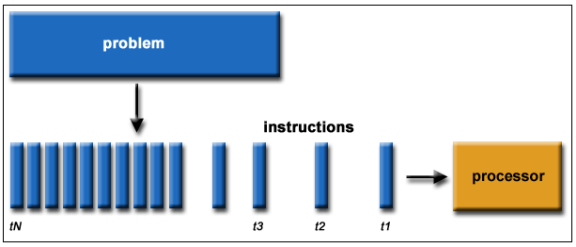


Figura 6: Computación serial [25]

Sin embargo, en la computación paralela la tarea se divide en el número de cores que tenga la máquina para realizar tareas en simultáneo y así ahorrar tiempo de ejecución (ver figura 7). Además, con paralelización es posible resolver problemas más complejos y al minimizar el tiempo (dependiendo de la aplicación) incluso podría ahorrarse algún costo [25].

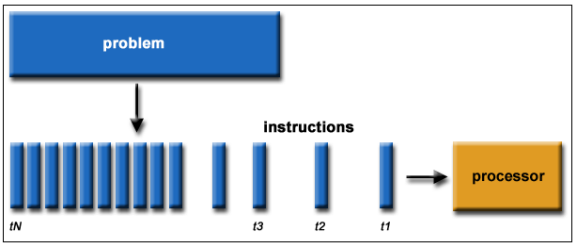


Figura 7: Computación paralela [25]

6.7.1. Programación paralela en Matlab

Matlab tiene un Toolbox de computación paralela con funciones como el *parfor*, que reemplaza el ciclo for normal por un ciclo for paralelizado. Para correr un programa paralelizado de Matlab primero es necesario habilitar el «Parallel Pool» de Matlab y establecer la cantidad de procesadores (Matlab les llama workers) a utilizar [25].

Para re-escribir un for normal a un *parfor* se requiere que cada una de las sub-tareas sea independiente, por lo que no pueden utilizarse entre sí. Además, también es necesario que el orden de ejecución de las tareas no importe (también el orden debe ser independiente). Algunas restricciones para el parfor son que no se puede introducir variables utilizando las funciones load, eval y global, no pueden contener break o return y tampoco puede contener

otro parfor dentro [25]. Para más información sobre este tema se recomienda ver [26].

Además de la programación paralela en Matlab, se recomienda utilizarla en conjunto con buenas prácticas de programación [27] y técnicas específicas para mejorar el desempeño [28], que incluyen estrategias como vectorización de código [29]. Si llegara a pasar que el código está trabajando de forma muy lenta es posible utilizar la técnica de búsqueda de cuellos de botella dada en [30] específicamente para Matlab utilizando la función *profiler*. Si se desea implementar paralelización en Matlab se recomienda leer todas las referencias bibliográficas de esta sección.

6.8. Planificación de movimiento

Se define como el problema de encontrar el movimiento desde un destino hasta un final esquivando obstáculos (si hubiese) y cumpliendo restricciones de juntas y/o torque para un robot [31].

6.8.1. Espacio de configuración

El espacio de configuración o espacio C corresponde a una configuración q del robot. Es decir, cada punto del espacio de configuración corresponde a una configuración diferente del robot. La configuración de un robot serial con n juntas está representada por el vector q con la posición de las n juntas, por lo que será un vector de dimensión n. El espacio libre en el espacio de configuración está dado por todas las configuraciones del robot donde este no colisiona con ningún obstáculo [31]. Este concepto de espacio de configuración se utilizará posteriormente para planificar la trayectoria.

6.8.2. Planificación de trayectorias

El problema de planificar una trayectoria es un sub-problema de la planificación de movimiento. En este caso se requiere encontrar un camino sin colisiones desde un inicio hasta un destino. Estos problemas pueden ser resueltos *online* u *offline*, dependiendo de si se necesita el resultado inmediatamente o si el ambiente es estático respectivamente [31].

6.9. Métodos de planificación de movimiento

6.9.1. Métodos completos

Los métodos completos se enfocan en representar de manera exacta la geometría del espacio libre de configuración [31].

Grafos de visibilidad

Este grafo es un mapa que utiliza los vértices de los obstáculos en el espacio de configuración como nodos. Estos se conectan solamente si pueden «*verse*». El peso de las aristas del grafo es la distancia euclidiana entre los nodos. Este método puede utilizarse y resultar adecuado para obstáculos poligonales [31].

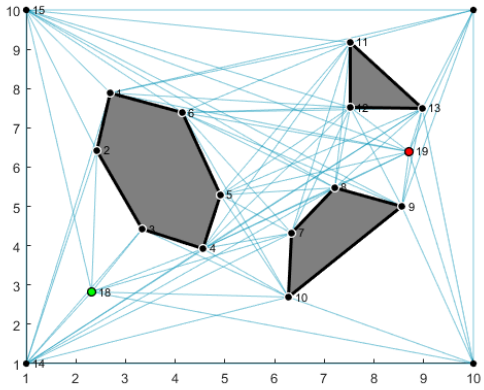


Figura 8: Ejemplo de grafo de visibilidad en Matlab.

6.9.2. Métodos de cuadrícula

Este método discretiza el espacio libre de configuración, creando una especie de cuadrícula para que el algoritmo busque una solución. Estos métodos pueden ser efectivos, pero la memoria requerida y el tiempo de búsqueda crecen de manera exponencial con el número de dimensiones del espacio. Es decir, una resolución de 100 en un espacio de dimensión 2 lleva a una cantidad de 100^2 nodos [31].

6.9.3. Métodos de muestreo

Estos métodos utilizan una función aleatoria o determinística para elegir muestras del espacio de configuración. Luego, evalúan si la muestra tomada está en el espacio libre y determina la muestra más cercana en el espacio libre para conectarlas. El resultado final de este algoritmo es un grafo o árbol que representa los movimientos legales del robot. Estos métodos sacrifican la resolución del mapa para minimizar la complejidad computacional [31].

Rapidly Exploring Random Trees (RRT)

Este método representa el espacio de configuración en forma de un árbol (grafo con nodos hijos), utilizando el método de muestreo aleatorio presentado anteriormente para encontrar el siguiente nodo disponible. Existe una variante de este método, donde se arma dos árboles: uno desde el nodo inicio y otro desde el nodo destino y cada cierto tiempo el algoritmo intenta unir ambos árboles.

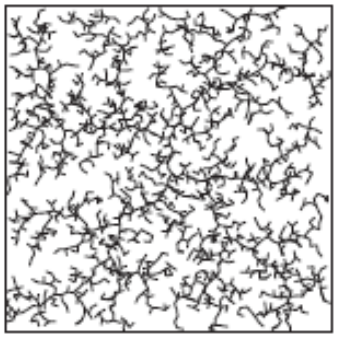


Figura 9: Ejemplo de un RRT [31]

Probability Road Maps (PRM)

Un PRM también representa el el espacio libre de configuración con puntos aleatorios conectados entre sí. Una ventaja de este algoritmo es que se puede crear de forma rápida, por lo que puede ser eficiente. La clave para el PRM es elegir cómo hacer el muestreo, pues generalmente se hace de forma aleatoria a partir de una distribución uniforme del espacio de configuración, pero se ha mostrado que los mejores resultados se obtienen al muestrear de forma más densa cerca de los obstáculos.

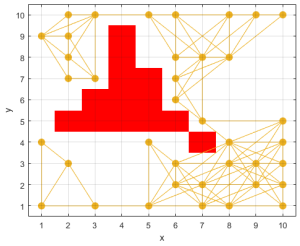


Figura 10: Ejemplo de un PRM en Matlab.

6.9.4. Virtual Potential Fields

Los campos de potencial virtuales crean fuerzas atractivas en el objetivo, atrayendo al robot en su dirección. Asimismo, crean fuerzas repulsivas en los obstáculos para alejar al robot de los mismos y así evitar colisiones. Este método es relativamente fácil de implementar y evaluar, pero puede quedarse atascado en mínimos locales de la función de potencial [31]. Este fue el método utilizado en la etapa II del proyecto Robotat UVG como puede observarse en [2].

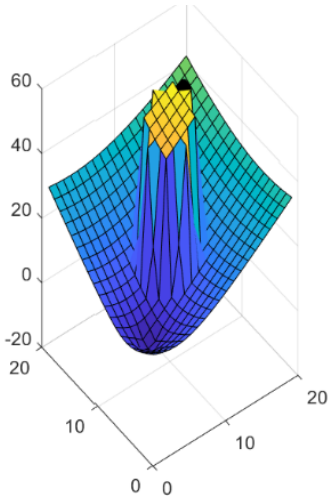


Figura 11: Ejemplo de un APF en Matlab.

6.10. Robots diferenciales

6.11. Controladores de posición y velocidad de robots diferenciales

CAPÍTULO 7

Diseño experimental AS

7.1. Experimento 1

Este experimento consiste principalmente de realizar una simulación simple del algoritmo Ant System. Primero se codificó el algoritmo Simple Ant Colony y luego se sobreescribió con el Ant System, que es un modelo un poco más complejo pues toma en consideración el costo por distancia y no solo la cantidad de feromona depositada. Además, este algoritmo también toma en cuenta otra variable para escalar la cantidad de feromona que se deposita. A continuación se presenta el pseudocódigo que se siguió (se recomienda ver el algoritmo 17.3 de [7]).

Algoritmo 7.1: Pseudocódigo de AS.

```
inicializar parametros
hasta que un porcentaje de las hormigas siga el mismo camino:
    por cada hormiga:
        hasta encontrar el nodo destino:
            caminar al siguiente nodo segun eq. de probabilidad
        end
    end
    por cada arista:
        evaporar feromona
        actualizar feromona segun largo de cada arista
    end
retornar el camino encontrado
```

Las variables en negrilla representan a variables tipo celda, de lo contrario la variable es

un array o matriz común y corriente. Asumiendo que tenemos k hormigas[1], n nodos en el grafo y un número máximo de iteraciones tf.

Primero presentaré a las variables con forma de matrices de adyacencia con pesos: mientras más grandes sean los valores, mayor probabilidad tendrán de ser escogidos por la función ant_decision.

$$\tau = \begin{bmatrix} \tau_{11} & \dots & \tau_{1n} \\ \vdots & \ddots & \vdots \\ \tau_{n1} & \dots & \tau_{nn} \end{bmatrix} \qquad \eta = \begin{bmatrix} \eta_{11} & \dots & \eta_{1n} \\ \vdots & \ddots & \vdots \\ \eta_{n1} & \dots & \eta_{nn} \end{bmatrix}$$

La siguiente variable es una celda que contiene vectores columna:

$$\boldsymbol{vytau} = \begin{bmatrix} \text{vecinos nodo 1} & \tau \text{ de los vecinos nodo 1} \\ \vdots & \vdots \\ \text{vecinos nodo n} & \tau \text{ de los vecinos nodo n} \end{bmatrix}$$

A partir de ahora utilizaremos la variable x para referirnos a vectores fila (x,y) apilados en una columna. En feas nodes se guardan los nodos no visitados o «viables» a los que cada nodo sí puede dirigirse. En la celda las node se guarda los nodos visitados anteriormente.

$$\boldsymbol{feas\_nodes} = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{k1} & \dots & x_{kn} \end{bmatrix} \qquad \boldsymbol{last\_node} = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{k1} & \dots & x_{kn} \end{bmatrix}$$

Blocked nodes guarda la lista de nodos ya visitados por cada nodo y path k guarda la lista de los nodos escogidos por la hormiga k (la actual).

$$\boldsymbol{blocked\_nodes} = \begin{bmatrix} x_{11} \\ \vdots \\ x_{k1} \end{bmatrix} \qquad \boldsymbol{path\_k} = \begin{bmatrix} x_{11} \\ \vdots \\ x_{k1} \end{bmatrix}$$

L guarda el costo total de cada path por hormiga (filas) y por iteración (columnas), mientras que all path guarda cada path (vectores fila apilados) por hormiga (filas) y por iteración (columnas).

$$L = \begin{bmatrix} L_{11} & \dots & L_{1t} \\ \vdots & \ddots & \vdots \\ L_{k1} & \dots & L_{kt} \end{bmatrix} \qquad \boldsymbol{last\_node} = \begin{bmatrix} x_{11} & \dots & x_{1t} \\ \vdots & \ddots & \vdots \\ x_{k1} & \dots & x_{kt} \end{bmatrix}$$

[1] en el código el número se expresa como «hormigas», pero por términos de simplificación se utilizará k en este documento

Parámetro	Valor
ρ	0.5
α	1.3
β	1
Q	1
tf	70
ϵ	0.9
τ_0	0.1

Cuadro 1: Parámetros del experimento 1.

Los parámetros son sensibles a la lejanía de los nodos, por lo que se recomienda hacer el barrido de parámetros con respecto a la lejanía con la que se desea probar el algoritmo. Para este experimento se utilizó el nodo (6,6) como nodo de prueba, utilizando los parámetros mostrados en el cuadro 1. En este experimento se realizó 10 corridas a mano, dado que mi computadora no puede correr muchos experimentos con 3 ciclos for anidados. Por lo mismo se propone que en un futuro se utilice una súper computadora como se explicará a continuación.

7.2. Experimento 2

Se notó que el algoritmo tardaba demasiado tiempo para hacer corridas de pruebas como barrido de parámetros (más de 8 horas para pocos parámetros en ciclos for anidados). Por lo tanto, se propuso realizar la implementación del algoritmo paralelizado, para correr el programa en la súper computadora del departamento de Ing. Electrónica UVG aprovechando todos sus cores. Para hacer el algoritmo paralelizado fue necesario modificar el ciclo for donde las hormigas recorren el mapa y encuentran un camino para que el proceso dentro fuera completamente independiente. Las variables que causaban problema eran feas_nodes y blocked_nodes, pero por la forma de implementarlo, no se podía modificar sin cambiar por completo el código.

Como se explicó en el marco teórico, Matlab cuenta con una clase grafo, por lo que se implementó el algoritmo haciendo uso de programación orientada a objetos. Los atributos de la clase grafo son Nodes y Edges, por lo que si creamos el grafo G, podemos acceder a esos atributos como sigue:

```
1 %Acceso a los atributos
2 grid_size = 10;
3 G = graph_grid(grid_size);
4 G.Nodes
5 G.Edges
```

Donde la función graph grid es una función auxiliar que se encarga de crear un grafo con los atributos Nodes y Edges, que son tablas con los siguientes parámetros:

```
% grid_graph.m
G = graph(table(EndNodes, Weight, Eta), table(Name, X, Y));
```

Como puede observarse, el primer argumento de la función graph corresponde al atributo Edge y el seguno al atributo Node. EndNodes representa dos nodos que conforman una arista, a la cual le corresponde un peso τ y un η, similar al código anterior. Con esto podemos eliminar las variables tau, eta y vytau. En el atributo Nodes tenemos el nombre del nodo y sus respectivas coordenadas.

Un objeto además de tener datos, tiene funciones, por lo que también contamos con los métodos neighbors y plot para encontrar a sus vecinos y visualizar el grafo respectivamente. En el código anterior, sin programación orientada a objetos, se tenía varias funciones como neighbors que ahora las eliminamos porque Matlab «ya hace el trabajo por nosotros».

Para sustituir el resto de variables, primero debemos darnos cuenta de que estas están relacionadas a cada hormiga. Por lo tanto, uno podría pensar que podemos crear un objeto hormiga y que cada una tenga sus atributos, pero debemos recordar que cada objeto debe tener datos y funciones. Entonces, en este caso no tenemos funciones por lo que opté por utilizar una estructura «ants» de tamaño igual a la cantidad de hormigas y con los siguientes parámetros:

```
hormigas = 50;
ants(1:hormigas) = struct('blocked_nodes', [], 'last_node',
    nodo_init, 'current_node', nodo_init, 'path', nodo_init, 'L',
    zeros(1, t_max));
```

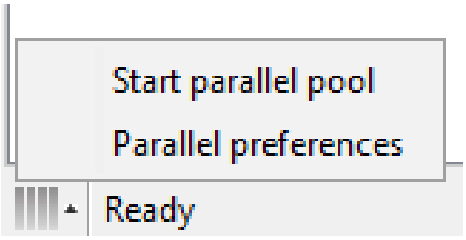
Esto se debe a que cada hormiga tendrá sus nodos bloqueados o lista tabú, su último nodo, su nodo actual, el path encontrado y la longitud del mismo. Aparte de estas variables también se tiene un histórico de los caminos y longitudes para comparar si los datos se están calculando de forma correcta.

Ya con estos cambios realizados es posible realizar la paralelización, donde cada iteración es completamente independiente de las demás. La paralelización en Matlab se activa cuando utilizamos la función parfor (parallel for) como se muestra a continuación:

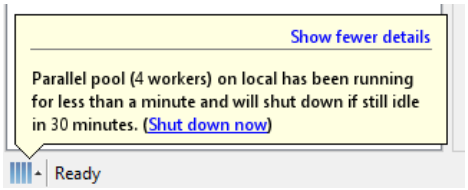
```
parfor k = 1:hormigas
    while(no se haya llegado al nodo destino)
        % construir el camino
    end
end
```

Para ejecutar el código es necesario prender lo que Matlab llama «Parallel Pool» en la esquina inferior izquierda (figura 12a). De no hacerlo, Matlab lo hace al ejecutar el código, pero se recomienda prenderlo antes para que sea más rápida la primera corrida.

El criterio de paro del algoritmo está dividido en dos partes: la primera detiene el algoritmo cuando la frecuencia de la moda corresponde a un ϵ en porcentaje (0-1) y el segundo



(a) Parallel pool sin empezar.



(b) Parallel pool lista para PC con 4 cores.

Figura 12: Parallel pool en Matlab.

criterio es un número máximo de iteraciones. Los parámetros utilizados en este experimento se muestran en 2.

Parámetro	Valor
ρ	0.5
α	1
β	1
Q	2
tf	70
ϵ	0.9
τ_0	1

Cuadro 2: Parámetros del experimento 2.

7.3. Experimento 3

Además de la representación del espacio como cuadrícula, también se incluyó la representación por medio de probabilistic road maps. Esta implementación se realizó utilizando la función del Toolbox de Robótica de Peter Corke, por lo que necesita instalarlo antes de correr el algoritmo. Para hacer varias corridas de esta representación se generó un grafo con la función prm_generator y se guardó en un archivo .mat, para luego cargarlo en Matlab y así hacer similares las corridas. Puede existir un problema con el Toolbox de Peter Corke con algunas funciones que él hizo que tienen el mismo nombre que otras funciones intrínsecas de Matlab. Este problema se resuelve y discute en el ReadMe del repositorio de GitHub de esta tesis.

El experimento de esta sección fue ejecutar el programa 10 veces a mano, al igual que en los experimentos anteriores. En este caso se utilizó los parámetros del cuadro 2, pero con una cantidad de hormigas igual a 100 (en los otros casos fue de 50) y un tf de 200 iteraciones.

7.4. Experimento 4

Experimentos con RRT

7.5. Experimento 5

Como extra se añadió la posibilidad de probar el Ant System con un espacio con obstáculos con un grafo de visibilidad. Sin embargo, esta implementación no considera la dimensión del robot, por lo que se asume que es una masa puntual y el espacio de trabajo es el mismo que el de configuración. Para una implementación más realista es necesario tomar en cuenta la dimensión del robot en los obstáculos y hacerlos más anchos.

El grafo de visibilidad se crea con una aplicación llamada poly_graph.mplapp que debe abrirse con el appdesigner de Matlab. Para editar el programa se ejecuta en la línea de comandos de matlab el comando: appdesigner y luego se abre el archivo. Para solamente ejecutar el programa y generar el grafo se puede dar doble click desde el «current folder» de Matlab. Primero se debe de presionar el botón de Add Obstacle y dibujar los polígonos que se desee, luego se presiona add start point para agregar un punto de inicio, add end point para agregar un punto final y finalmente visibility graph para generar el grafo y guardarlo en un archivo .mat en el folder donde está actualmente. En el archivo principal de ACO se importa el último archivo generado.

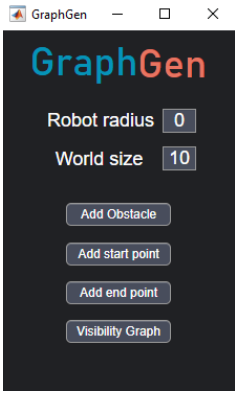


Figura 13: Aplicación que genera grafos de visibilidad.

Se realizó la interfaz de la aplicación con dibujos personalizados por el usuario para que en el futuro cuando se quiera probar con una cámara se pueda hacer el trazo a mano sobre la imagen o que incluso con el mismo Matlab se procese la imagen con visión por computadora. Las funciones de Matlab que hacen esto son similares a la utilizada ahora, por lo que en un futuro no debería de ser tan difícil la implementación. La aplicación también cuenta con un espacio para colocar el radio del robot para que en el futuro se pueda implementar con restricciones físicas.

El experimentó consistió en ejecutar el algoritmo 10 veces, al igual que en los experimentos anteriores, con 50 hormigas y los parámetros mostrados en el cuadro 2, exceptuando tf que fue de 200.

CAPÍTULO 8

Resultados AS

8.1. Experimento 1

A continuación se muestra un camino óptimo y una imagen de su simulación de feromona hallada por el programa.

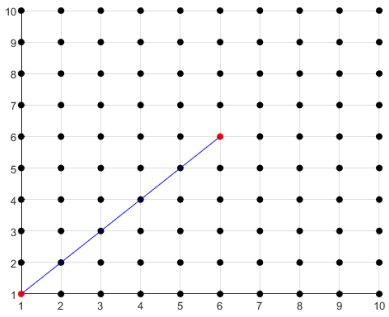


Figura 14: Camino óptimo encontrado del nodo (1,1) al (6,6).

Los resultados se muestran en el cuadro 3.

En este ejemplo el costo mínimo que se podía obtener era 2.5, ya que el costo de las diagonales era de 0.5. El tiempo fue medido con las funciones tic y toc de Matlab. Al final de la tabla se muestra la media de cada parámetro. Para cada corrida es posible observar que el tiempo aumenta cuando las hormigas toman malas decisiones al principio. Puede que esto se arregle modificando la tasa de evaporación para aumentar el tiempo de exploración de las hormigas.

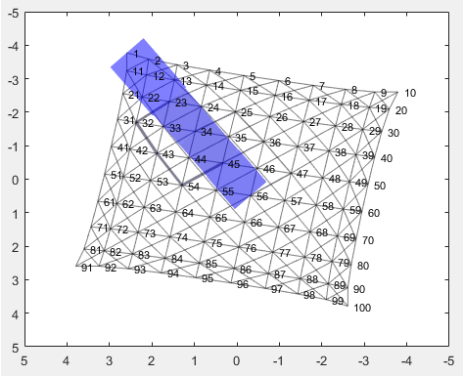


Figura 15: Feromona del camino óptimo encontrado del nodo (1,1) al (6,6).

t	tiempo	costo
14	14.2161	2.5
10	9.6620	2.5
9	9.2031	2.5
18	49.5735	3.5
43	44.7586	3.5
17	16.0491	2.5
26	25.3741	3.5
11	26.8102	2.5
22	60.7632	3.5
13	12.3290	3.5
18.3	**26.8739**	**3**

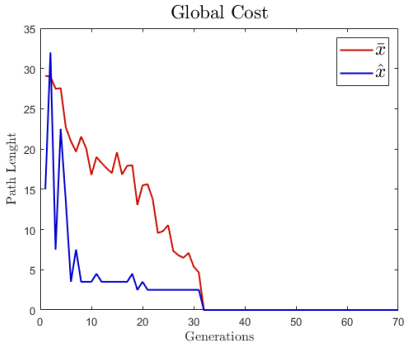
Cuadro 3: Resultados del experimento 1.

8.2. Experimento 2

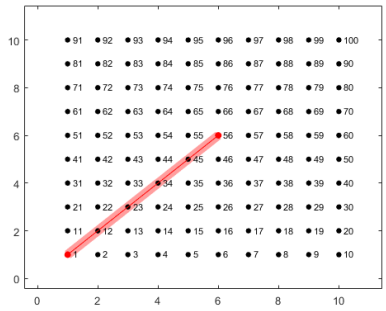
Al ejecutarlo veremos dos gráficas animadas: la figura 1 tendrá la media $\bar{x}$ del largo del camino y la moda $\hat{x}$ del mismo, mientras que la segunda tiene una animación de los caminos y su feromona. Esta última animación cambia de color y de grosor de línea en cada arista que tiene mayor τ. Dentro del código se agrega un nuevo parámetro al atributo Edges que guarda la cantidad de feromona normalizada para utilizarla en esta animación. El color rojo indica mayor presencia de feromona, mientras que el color blanco indica que esa arista tiene menor cantidad.

Los resultados se muestran en el cuadro 4.

[MOSTRAR TABLAS COMPARANDO TIEMPOS DE EJECUCIÓN CON Y SIN PARALELIZACIÓN]



(a) Media y moda del largo de los caminos.

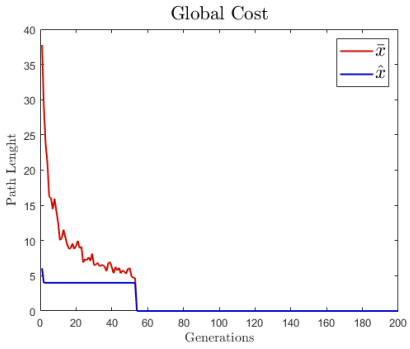


(b) Animación de feromona y camino encontrado.

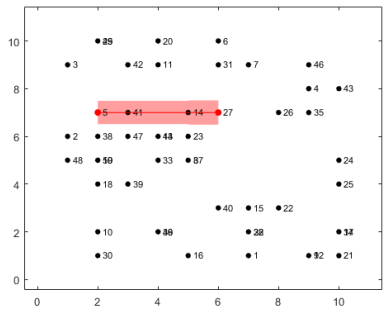
Figura 16: Animaciones generadas con mundo cuadriculado.

t	tiempo	costo
30	47.72	2.5
33	56.22	2.5
27	46.65	2.5
54	97.47	3.5
24	43.62	2.5
25	44.37	2.5
53	96.31	2.5
70	125.35	2.5
42	69.83	2.5
42	70.58	2.5
40	**69.81**	**2.6**

Cuadro 4: Resultados del experimento 2.



(a) Media y moda del largo de los caminos.



(b) Animación de feromona y camino encontrado.

Figura 17: Animaciones generadas con PRM.

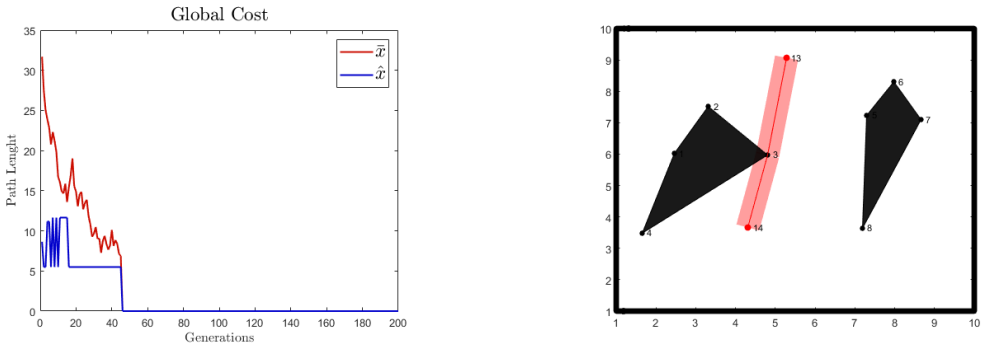
t	tiempo	costo
69	79.30	4.00
146	152.03	6.00
105	117.28	4.00
82	88.59	4.00
83	87.08	6.00
192	211.31	6.00
178	206.50	6.00
95	106.60	6.00
117	130.32	6.00
53	53.20	4.00
112	**123.22**	**5.2**

Cuadro 5: Resultados del experimento 3.

8.3. Experimento 3

8.4. Experimento 4

8.5. Experimento 5



(a) Media y moda del largo de los caminos.

(b) Animación de feromona y camino encontrado.

Figura 18: Animaciones generadas con grafo de visibilidad.

t	tiempo	costo
56	23.46	7.88
114	42.96	7.88
41	17.48	5.49
129	47.03	7.88
100	38.04	7.88
82	30.19	5.49
36	21.72	7.88
172	65.71	7.88
45	31.17	5.49
87	32.18	5.49
86.2	**34.99**	**6.92**

Cuadro 6: Resultados del experimento 5.

CAPÍTULO 9

Diseño experimental GA

CAPÍTULO 10

Resultados GA

CAPÍTULO 11

Conclusiones

CAPÍTULO 12

Recomendaciones

Recomiendo abrir una línea de investigación sobre Ant algorithms para aplicaciones de VLSI

El siguiente que haga esto pero con obstáculos

Comparar esto pero con métodos tradicionales como A*, Dijkstra, Breathfire.

HAcer mezcla entre algoritmos genéticos y pso o algoritmos genéticos y aco.

Pasarlo a python y ver si es más rápido

Cuando ya esté todo listo pueden probarlo en el robotarium de Georgia Tech

Mezclar esto pero con Qlearning para los parámetros quizás.

Probar otros tipos de ant colony como maxmin y antQ.

Probar otros tipos de cruza y mutación en algoritmos genéticos.

Usar algoritmos genéticos combinados con AFP.

Se podrá resolver este problema con grafos y el PSO?

CAPÍTULO 13

Bibliografía

[1] A. S. A. Nadalini, «Algoritmo Modificado de Optimización de Enjambre de Partículas (MPSO)», Universidad del Valle de Guatemala, UVG, nov. de 2019.

[2] J. P. C. Pérez, «Implementación de enjambre de robots en operaciones de búsqueda y rescate», Universidad del Valle de Guatemala, UVG, nov. de 2019.

[3] *Robotarium | Institute for Robotics and Intelligent Machines*. dirección: `http://www.robotics.gatech.edu/robotarium`.

[4] *Programmable Robot Swarms*, en-US, ago. de 2016. dirección: `https://wyss.harvard.edu/technology/programmable-robot-swarms/` (visitado 06-04-2020).

[5] S. P. ROUL, «Application of Ant Colony Optimization for finding Navigational Path of mobile robot», National Institute of Technology, Rourkela, 2011.

[6] *Comparative Analysis of Ant Colony and Particle Swarm Optimization Techniques V.Selvi Lecturer, Department of Computer Science, Nehru Memorial College,*

[7] A. P. Engelbrecht, *Computational intelligence: an introduction*. Wiley, 2008.

[8] M. N. A. Wahab, S. Nefti-Meziani y A. Atyabi, «A Comprehensive Review of Swarm Optimization Algorithms», *PLoS ONE*, vol. 10, n.º 5, 2015. DOI: `https://doi.org/10.1371/journal.pone.0122827`.

[9] Y. Zhang, S. Wang y G. Ji, «A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications», *Hindawi*, vol. 2015, n.º 931256, 2015. DOI: `http://dx.doi.org/10.1155/2015/931256`.

[10] M. Dorigo y T. Stützle, *Ant colony optimization*, en. Cambridge, Mass: MIT Press, 2004, ISBN: 978-0-262-04219-2.

[11] U. Sydney, *Genetic Algorithm:A Learning Experience*. dirección: `http://www.cse.unsw.edu.au/~cs9417ml/GA2/encoding_other.html` (visitado 27-07-2020).

[12] D. Samanta, «Encoding Techniques in Genetic Algorithms», en, pág. 42,

[13] M. Obitko, *Encoding - Introduction to Genetic Algorithms - Tutorial with Interactive Java Applets*, 1998. dirección: `https://www.obitko.com/tutorials/genetic-algorithms/encoding.php` (visitado 27-07-2020).

[14] D. Bingham. (2017). Optimization Test Problems, dirección: https://www.sfu.ca/~ssurjano/optimization.html.

[15] P. L. Aga, C. M. H. Kuijpers, R. H. Murga, I. Inza y S. Dizdarevic, «Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators», en, pág. 42,

[16] T. Blickle y L. Thiele, «A Mathematical Analysis of Tournament Selection», en, pág. 8,

[17] H. Mühlenbein y D. Schlierkamp-Voosen, «Predictive Models for the Breeder Genetic Algorithm I. Continuous Parameter Optimization», en, *Evolutionary Computation*, vol. 1, n.º 1, págs. 25-49, mar. de 1993, ISSN: 1063-6560, 1530-9304. DOI: 10.1162/evco.1993.1.1.25. dirección: http://www.mitpressjournals.org/doi/10.1162/evco.1993.1.1.25 (visitado 27-07-2020).

[18] K. Rodríguez Vázquez, *Cómputo evolutivo - Inicio*, es, Library Catalog: www.coursera.org. dirección: https://www.coursera.org/learn/computo-evolutivo/home/welcome (visitado 27-07-2020).

[19] C. A. Maeso, *Métodos basados en grafos*. dirección: http://pdg.cnb.uam.es/pazos/cursos/bionet_UAM/Grafos_CAguirre.pdf.

[20] K. Thulasiraman y M. N. S. Swamy, *Graphs: Theory and Algorithms*, en. John Wiley & Sons, mar. de 2011, Google-Books-ID: rFH7eQffQNkC, ISBN: 978-1-118-03025-7.

[21] K. " Thulasiraman, S. Arumugam, A. Brandstädt y T. Nishizeki, eds., *Handbook of Graph Theory, Combinatorial Optimization, and Algorithms*, en, 0.ª ed. Chapman y Hall/CRC, ene. de 2016, ISBN: 978-0-429-15023-4. DOI: 10.1201/b19163. dirección: https://www.taylorfrancis.com/books/9781420011074 (visitado 12-07-2020).

[22] *Graph with undirected edges - MATLAB - MathWorks América Latina*. dirección: https://la.mathworks.com/help/matlab/ref/graph.html (visitado 12-07-2020).

[23] D. Phillips, *Python 3 object oriented programming: harness the power of Python 3 objects*, en, ép. Community experience distilled. Birmingham: Packt Publ, 2010, OCLC: 802342008, ISBN: 978-1-84951-126-1.

[24] *Clases - MATLAB & Simulink - MathWorks América Latina*. dirección: https://la.mathworks.com/help/matlab/object-oriented-programming.html (visitado 12-07-2020).

[25] T. Mathieu, G. Hernandez y A. Gupta, «Parallel Computing with MATLAB», en, pág. 56,

[26] *Using parfor Loops: Getting Up and Running*, Library Catalog: blogs.mathworks.com. dirección: https://blogs.mathworks.com/loren/2009/10/02/using-parfor-loops-getting-up-and-running/ (visitado 13-07-2020).

[27] *Programming Patterns: Maximizing Code Performance by Optimizing Memory Access*, en, Library Catalog: la.mathworks.com. dirección: https://la.mathworks.com/company/newsletters/articles/programming-patterns-maximizing-code-performance-by-optimizing-memory-access.html (visitado 13-07-2020).

[28] *Techniques to Improve Performance - MATLAB & Simulink - MathWorks América Latina*. dirección: https://la.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html (visitado 13-07-2020).

[29] *Vectorization - MATLAB & Simulink - MathWorks América Latina*. dirección: https://la.mathworks.com/help/matlab/matlab_prog/vectorization.html (visitado 13-07-2020).

[30] *Buscar cuellos de botella de código - MATLAB & Simulink - MathWorks América Latina*. dirección: https://la.mathworks.com/help/matlab/creating_plots/assessing-performance.html (visitado 13-07-2020).

[31] K. M. Lynch y F. C. Park, *Modern robotics: mechanics, planning, and control*, en. Cambridge, UK: Cambridge University Press, 2017, OCLC: ocn983881868, ISBN: 978-1-107-15630-2 978-1-316-60984-2.

CAPÍTULO 14

Anexos

14.1. Repositorio de Github

CAPÍTULO 15

Glosario

fórmula Una expresión matemática. 9

latex Es un lenguaje de marcado adecuado especialmente para la creación de documentos científicos. 9