
Optimization of AI algorithms by applying high-performance
computing techniques

Optimización de algoritmos de IA aplicando técnicas enfocadas al
cómputo de alto rendimiento



TRABAJO DE FIN DE GRADO

Grado en Ingeniería Informática

DANIEL PIZARRO GALLEG0

Director:

Alberto Núñez Covarrubias

Facultad de Informática
Universidad Complutense de Madrid

13 de septiembre del 2024

Autorización de difusión

Autor

Daniel Pizarro Gallego

Fecha

Madrid, 11 de Septiembre de 2024.

El abajo firmante, matriculado en el Grado de Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado: Optimización de algoritmos de IA aplicando técnicas enfocadas en cómputo de alto rendimiento, realizado durante el curso académico 2023-2024 bajo la dirección de Alberto Núñez Covarrubias en el Departamento de Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

El trabajo que se presenta se enfoca en la optimización de algoritmos de Inteligencia Artificial (IA) mediante el uso de MPI (Message Passing Interface), una biblioteca estándar desarrollada para el cómputo de alto rendimiento. El objetivo principal consiste en reducir el tiempo de ejecución de los algoritmos, explotando el paralelismo de los recursos de cómputo y la memoria distribuida. Esta tarea es especialmente relevante debido al alto coste computacional y de recursos que implica entrenar o ejecutar estos algoritmos.

Este proyecto incluye una descripción de los fundamentos teóricos de los algoritmos que se van a implementar, así como el funcionamiento de la biblioteca MPI. Una vez puesto en contexto, se desarrollan en profundidad las estrategias propuestas para mejorar los algoritmos. Además, se ha realizado un exhaustivo estudio empírico para analizar las estrategias desarrolladas, las cuales han sido ejecutadas en un ordenador personal y en un sistema distribuido que consta de 128 núcleos de CPU y 256 GB de RAM.

Palabras clave

IA, aprendizaje automático, MPI, speed-up, distribuida, redes neuronales, algoritmos, clustering, master, worker.

Abstract

The work presented focuses on the optimization of Artificial Intelligence (AI) algorithms using MPI (Message Passing Interface), a standard library developed for high-performance computing. The main objective consists in reducing the execution time of the algorithms, by exploiting the parallelism of computing resources and distributed memory. This task is especially relevant due to the high computational and resource cost involved in training or running these algorithms. This project includes a description of the theoretical foundations of the algorithms that will be implemented. Moreover, functioning of the MPI library is also presented. Once put in context, the strategies employed to enhance the algorithms are described in detail. In addition, an exhaustive empirical study has been carried out to analyze the developed strategies, which have been executed on a personal computer and in a high distributed system consisting of 128 CPU cores and 256 GB of RAM.

Keywords

IA, machine learning, MPI, speed-up, distributed, neural network, algorithm, clustering, master, worker.

Índice general

Índice	I
Dedicatoria	III
1. Introducción	1
1.1. Definición y alcance del proyecto	1
1.2. Motivación	3
1.3. Objetivo	4
1.4. Estructura del documento	5
1. Introduction	6
1.1. Project definition and scope	6
1.2. Motivation	8
1.3. Objective	8
1.4. Document structure	10
2. Contextualización	11
2.1. MPI	11
2.2. Aprendizaje por Refuerzo	14
2.2.1. Algoritmo Q-Learning	14
2.2.2. Deep Q-Network (DQN)	16
2.3. Aprendizaje No-Supervisado	18
2.3.1. Clustering jerárquico aglomerativo	19
2.3.2. Clustering basado en particiones: K-Medias	20
2.4. Aprendizaje Supervisado	22
2.4.1. K-Vecinos más Cercanos - KNN	22
2.4.2. Redes Neuronales	23
2.5. Algoritmos Evolutivos	25
3. Diseño e Implementación de estrategias para aumentar el rendimiento de algoritmos de IA	27
3.1. Programas sencillos	27
3.2. Algoritmos de Clustering	32
3.2.1. Jerárquico Aglomerativo	33
3.2.2. K-Medias	36
3.2.3. K-Vecinos más cercanos (KNN)	39
3.3. Aprendizaje por refuerzo	41

3.3.1.	Q-Learning	42
3.3.2.	Deep Q-Network	46
3.4.	Algoritmos Evolutivos	49
3.5.	Redes Neuronales	56
4.	Estudio empírico	61
4.1.	Entornos de ejecución	61
4.2.	Programas sencillos	63
4.2.1.	Ordenaciones	63
4.2.1.1.	Algoritmos de complejidad cuadrática	63
4.2.1.2.	Algoritmo <i>MergeSort</i>	65
4.2.2.	Multiplicación de matrices	67
4.3.	Algoritmos de Agrupación	69
4.3.1.	Jerárquico Aglomerativo	71
4.3.2.	K-Medias	75
4.3.3.	KNN	78
4.4.	Q-Learning	81
4.5.	Algoritmos Evolutivos	85
4.6.	Redes Neuronales	96
5.	Conclusiones y trabajo futuro	102
5.	Conclusions and future work	104
	Bibliography	108

Dedicatoria

*A mis padres, por que gracias a ellos soy quien
soy hoy*

Capítulo 1

Introducción

En este capítulo se presenta una perspectiva general del contexto en el que se ha llevado a cabo el proyecto. Además de las dificultades encontradas durante su desarrollo para alcanzar las contribuciones mencionadas, se detallan cada uno de los propósitos perseguidos en él.

1.1. Definición y alcance del proyecto

El desarrollo de las Inteligencias Artificiales en nuestra sociedad ha sido un fenómeno de gran relevancia, además de popular, en los últimos años. Estas tecnologías han llegado para quedarse y están mejorando nuestra calidad de vida. Desde la automatización de tareas hasta la asistencia virtual¹², estas IAs desempeñan un papel cada vez más importante en nuestro día a día. Con el advenimiento del Internet de alta velocidad y la proliferación de datos, las empresas tecnológicas se enfrentan a la necesidad creciente de desarrollar servicios de alta calidad en un mercado muy competitivo. Actualmente, se invierte mucho dinero y tiempo en mejorar y diseñar algoritmos para implementar Inteligencias Artificiales para el acceso público²¹.

El entrenamiento y ejecución de estos algoritmos para modelar inteligencias artificiales consumen mucha energía, además de provocar una cantidad excesiva de emisiones de CO₂. La empresa tecnológica *Hugging Face*, desarrolladora de *BLOOM*¹, la primera LLM (Large Language Model) multilinguaje entrenada de forma transparente, tuvo la colaboración de muchos investigadores. Este proyecto, con 176 mil millones de parámetros, es capaz de

generar texto en 46 idiomas y 13 lenguajes de programación. No obstante, estimaron que el entrenamiento de esta inteligencia artificial emitió 25 toneladas de CO₂, cifra que se duplicó al contar el coste de producción del equipo informático usado¹⁰. Los investigadores se están enfocando en evaluar y reducir el impacto ambiental de las tecnologías de IA. Una prueba de ello es el desarrollo de *CarbonTracker*⁹ (CTI), un equipo de especialistas financieros que asumen el riesgo climático como realidad de los mercados financieros actuales. El objetivo de esta herramienta es predecir y reducir la huella de carbono de las etapas de entrenamiento de los modelos de IA¹⁵.

El uso de la programación distribuida, más específicamente, aplicaciones basadas en MPI, permite ejecutar varios procesos en paralelo, dividiendo la carga de trabajo y reduciendo el tiempo de ejecución. Al contrario de los programas basados en el modelo de memoria compartida, en el cual se pueden dar problemas de sincronización, cada proceso generado tiene su propia memoria local, evadiendo estos problemas. Sin embargo, hay que diseñar implementaciones correctas y eficientes para no tener más complejidad espacial (uso de memoria) de la esperada. Las conexiones entre los procesos se pueden configurar para maximizar la eficiencia y reducir el tiempo de cómputo. Una de las más populares es el modelo *Master-Worker*. El proceso *master* se encarga de distribuir el trabajo a los procesos *workers* para que, en paralelo, puedan ejecutar la misma tarea con conjuntos de datos más reducidos. Al finalizar la tarea, cada *worker* envía sus datos procesados, y si el proceso *master* no ha terminado la ejecución, espera para recibir más datos hasta procesar completamente el data-set inicial. MPI permite la comunicación eficiente entre procesos, mejorando la escalabilidad y reduciendo el tiempo de procesamiento. Esta metodología, además de mejorar el rendimiento, también simplifica la gestión de recursos, mejorando la utilización del hardware disponible en el sistema.

Los algoritmos de IA suelen manejar un vasto número de datos para entrenar y evaluar los modelos deseados. Por eso es fundamental diseñar estrategias para distribuir los datos y dividir las cargas de trabajo de manera equitativa, controlando el flujo de datos para

evitar cuellos de botella (un proceso consume más tiempo de ejecución que los demás, quedando el resto en espera), y equilibrar los recursos disponibles. Esto incluye, además de la optimización del tiempo de ejecución, una gestión efectiva de la memoria y una reducción en la latencia (tiempo de espera en transmitir los paquetes de información en una red) en la comunicación entre los procesos.

En este proyecto se diseñarán e implementarán varias estrategias para diferentes algoritmos de IA, con el objetivo de reducir el tiempo de ejecución.

1.2. Motivación

Actualmente hay muchas implementaciones de algoritmos de IA. *Scikit learn* es una biblioteca de *Python* adecuada para probar cualquier técnica. Esta biblioteca, como la mayoría, ejecuta los algoritmos de manera secuencial, sin dividir la carga de trabajo. Las implementaciones están estudiadas y perfeccionadas para realizar los cálculos en un único proceso, pero se puede reducir el tiempo de ejecución aplicando paralelismo.

Las arquitecturas distribuidas, junto con las técnicas de cómputo de alto rendimiento (HPC, por sus siglas en inglés), son una de las soluciones más apropiadas para los usuarios finales: científicos y empresas. La búsqueda de un rendimiento óptimo, enfocada a sistemas altamente distribuidos, demandan enfoques adaptables y escalables para implementar aplicaciones científicas de alto rendimiento. Sincronizar los procesos, distribuir los datos para aumentar el paralelismo y reducir el *overhead* (coste adicional asociado a la gestión de los recursos y la comunicación entre procesos), son los desafíos recurrentes en los sistemas distribuidos. La comunicación tiene que ser controlada para el correcto funcionamiento y, en algunas ocasiones, estas mejoras derivan en implementaciones más complejas.

Generalmente, en estos casos, es desafiante controlar las acciones de cada proceso para obtener la especificación deseada. Las empresas tecnológicas buscan mejorar el rendimiento y reducir costes de sus sistemas. Para ello, es necesario un uso eficiente de los recursos computacionales. Además, existe un interés en reducir el consumo de energía y las emisiones

de CO₂, al entrenar o procesar modelos de IA, puesto que el impacto ambiental está en auge hoy en día. Contando con una gran competitividad en el mercado tecnológico, cualquier mejora, aunque excesivamente significativa, es un avance.

1.3. Objetivo

El objetivo principal de este trabajo es **paralelizar algoritmos de IA, empleando técnicas de HPC para reducir el tiempo de ejecución**. Entre los algoritmos a optimizar se encuentran técnicas como el agrupamiento de individuos, predicción de resultados y la optimización de funciones de evaluación. Todas estas implementaciones han sido desarrolladas en *Python*, el lenguaje de programación más popular -actualmente- en el ámbito de la inteligencia artificial¹⁹. Sin embargo, al ser un lenguaje interpretado (el código se traduce en la misma ejecución), aumenta la sobrecarga y hace que -generalmente- el programa sea más lento que la misma implementación en otros lenguajes. Por estos motivos, *Python* es el lenguaje de programación idóneo para aplicar técnicas de cómputo de alto rendimiento y reducir el tiempo de ejecución.

Asimismo, se requerirá alcanzar los siguientes objetivos secundarios:

1. **Diseño de implementaciones escalables y flexibles.** Al diseñar e implementar mejoras de cada algoritmo, es posible utilizar -entre otros parámetros- distintos números de procesos en la ejecución. Esto permite un estudio profundo de las implementaciones realizadas. Al variar el número de procesos se puede comprobar cuál es el número idóneo para cualquier implementación. La flexibilidad en las mejoras permite variar los datos de entrada para que funcione correctamente con un tamaño de *dataset* variable.
2. **Correcto funcionamiento de los algoritmos.** Al realizar las mejoras, además de mejorar el rendimiento, es necesario tener una cohesión con el algoritmo original. Es decir, si queremos maximizar una función de evaluación, la implementación de la mejo-

ra tiene que proporcionar resultados parecidos o mejores. No resulta útil implementar una mejora que reduzca el tiempo de ejecución, pero que obtenga peores resultados que el algoritmo ejecutado secuencialmente.

3. **Estudio empírico.** Se realizará una evaluación de las mejoras para calcular el aumento del rendimiento. Primero, se mide el tiempo de ejecución de cada algoritmo sin mejoras y, posteriormente, se analizan las diferentes implementaciones desarrolladas variando el número de procesos ejecutados. Para finalizar, se prueban las mejores implementaciones de cada algoritmo en un ordenador personal y en un sistema distribuido con *128* núcleos.

1.4. Estructura del documento

El resto de este documento está organizado en los siguientes capítulos:

- Capítulo 2: Contextualización. En este capítulo se proporciona información de cada algoritmo estudiado.
- Capítulo 3: Diseño e implementaciones. Este capítulo describe en detalle las implementaciones desarrolladas para las diferentes técnicas abordadas.
- Capítulo 4: Estudio empírico, presenta el proceso experimental realizado, el cual consiste en analizar las mejoras propuestas, variando el número de procesos y el tamaño de los datos, en dos entornos distintos, ordenador personal y sistema distribuido.
- Capítulo 5: Conclusiones y trabajo a futuro. El último capítulo finaliza el trabajo con una síntesis de los resultados obtenidos, y presenta la proyección del trabajo a futuro.

Todo el trabajo realizado para este proyecto (código, *datasets*, pruebas, archivos photoshop, etc) está almacenado en el siguiente repositorio de GitHub [Danipiza/TFG](#). Las imágenes utilizadas en la memoria han sido creadas desde cero en Photoshop.

Introduction

This chapter presents a general perspective of the context in which the project has been carried out. As well as the difficulties encountered during its development to achieve the aforementioned contributions, each of the purposes pursued in it are detailed.

1.1. Project definition and scope

The development of Artificial Intelligence in our society has been a phenomenon of great relevance, as well as popular, in recent years. These technologies are here to stay and are improving our quality of life. From task automation to virtual assistance¹², these AIs play an increasingly important role in our daily lives. With the advent of high-speed Internet and the proliferation of data, technology companies are faced with the growing need to develop high-quality services in a highly competitive market. Currently, a lot of money and time is invested in improving and designing algorithms to implement Artificial Intelligence for public access²¹.

The training and execution of these algorithms to model artificial intelligence consume a lot of energy, in addition to causing an excessive amount of CO₂ emissions. The technology company *Hugging Face*, developer of *BLOOM*¹, the first transparently trained multilanguage LLM (Large Language Model), had the collaboration of many researchers. This project, with 176 billion parameters, is capable of generating text in 46 languages and 13 programming languages. However, they estimated that training this artificial intelligence emitted 25 tons of CO₂, which is doubled when counting the production cost of the computer equipment used¹⁰. Researchers are focusing on evaluating and reducing the environmental impact of AI technologies. Proof of this is the development of *CarbonTracker*⁹ (CTI), a team of financial specialists who assume climate risk as a reality of current financial markets. The objective of

this tool is to predict and reduce the carbon footprint of the training stages of AI¹⁵ models.

The use of distributed programming, more specifically, MPI-based applications, allows multiple processes to run in parallel, splitting the workload and reducing execution time. Unlike programs based on the shared memory model, in which synchronization problems can occur, each generated process has its own local memory, avoiding these problems. However, correct and efficient implementations must be designed to avoid having more spatial complexity (memory usage) than expected. Connections between processes can be configured to maximize efficiency and reduce computing time. One of the most popular is the *Master-Worker* model. The *master* process is responsible for distributing the work to the *workers* processes so that, in parallel, they can execute the same task with smaller data sets. At the end of the task, each *worker* sends its processed data, and if the *master* process has not finished the execution, it waits to receive more data until it completely processes the initial data-set. MPI enables efficient communication between processes, improving scalability and reducing processing time. This methodology, in addition to improving performance, also simplifies resource management, improving the utilization of the hardware available in the system.

AI algorithms typically handle vast numbers of data to train and evaluate desired models. That is why it is essential to design strategies to distribute data and divide workloads equitably, controlling the flow of data to avoid bottlenecks (one process consumes more execution time than the others, leaving the rest waiting), and balance available resources. This includes, in addition to optimization of execution time, effective memory management and a reduction in latency (waiting time for transmitting information packets on a network) in communication between processes.

In this project, several strategies will be designed and implemented for different AI algorithms, with the aim of reducing execution time.

1.2. Motivation

There are currently many implementations of AI algorithms. *Scikit learn* is a *Python* library suitable for testing any technique. This library, like most, runs the algorithms sequentially, without splitting the workload. The implementations are studied and perfected to perform the calculations in a single process, but the execution time can be reduced by applying parallelism.

Distributed architectures, together with high-performance computing (HPC) techniques, are one of the most appropriate solutions for end users: scientists and companies. The search for optimal performance, focused on distributed systems, demands adaptive and scalable approaches to implement high-performance scientific applications. Synchronizing processes, distributing data to increase parallelism and reduce *overhead* (additional cost associated with resource management and communication between processes), are recurring challenges in distributed systems. Communication has to be controlled for correct operation and, on some occasions, these improvements lead to more complex implementations.

Generally, in these cases, it is challenging to control the actions of each process to obtain the desired specification. Technology companies seek to improve the performance and reduce costs of their systems. For this, efficient use of computational resources is necessary. In addition, there is an interest in reducing energy consumption and CO₂ emissions, when training or processing AI models, since the environmental impact is on the rise today. With great competitiveness in the technological market, any improvement, even if excessively significant, is an advance.

1.3. Objective

The main objective of this work is **parallelize AI algorithms, using HPC techniques to reduce execution time**. Among the algorithms to be optimized are techniques such as clustering of individuals, prediction of results and optimization of evaluation functions. All of these implementations have been developed in *Python*, the most popular programming

language -currently- in the field of artificial intelligence¹⁹. However, being an interpreted language (the code is translated in the same execution), it increases the overhead and makes the program -generally- slower than the same implementation in other languages. For these reasons, *Python* is the ideal programming language to apply high-performance computing techniques and reduce execution time.

Likewise, it will be necessary to achieve the following secondary objectives:

1. **Design of scalable and flexible implementations.** When designing and implementing improvements to each algorithm, it is possible to use -among other parameters- different numbers of processes in the execution. This allows a deep study of the created implementations. By varying the number of processes you can check which number is ideal for any implementation. Flexibility in the enhancements allows the input data to be varied to work correctly with a variable *dataset* size.
2. **Correct operation of the algorithms.** When making improvements, in addition to improving performance, it is necessary to have cohesion with the original algorithm. That is, if we want to maximize an evaluation function, the implementation of the improvement has to provide similar or better results. It is not useful to implement an improvement that reduces the execution time, but obtains worse results than the algorithm executed sequentially.
3. **Empirical study.** An evaluation of the improvements will be carried out to calculate the obtained speed-up. First, the execution time of each algorithm is measured without improvements and, subsequently, the different implementations developed are analyzed by varying the number of processes executed. Finally, the best implementations of each algorithm are tested on a personal computer and on a distributed system consisting of 128 cores.

1.4. Document structure

The subsequent chapters of this document are organized as follows:

- Chapter 2: Contextualization. This chapter provides information on each algorithm studied.
- Chapter 3: Design and implementations. This chapter describes in detail the implementations developed for the different techniques addressed.
- Chapter 4: Empirical study, presents the experimental process carried out, which consists of analyzing the proposed improvements, varying the number of processes and the size of the data, in two different environments, personal computer and distributed system.
- Chapter 5: Conclusions and future work. The last chapter concludes the work with a synthesis of the results obtained, and presents the projection of future work.

The material generated in this work (code, datasets, tests, photoshop files, etc.) is stored in the following GitHub repository [Danipiza/TFG](#). The images used in this paper have been created from scratch in Photoshop.

Capítulo 2

Contextualización

En este capítulo se presenta una descripción de los algoritmos de Inteligencia Artificial que se van a utilizar en el proyecto, profundizando en sus usos y características. Además, se presenta la biblioteca de paso de mensajes (MPI) empleada en el proyecto para la paralelización de los algoritmos.

2.1. MPI

Message Passing Interface⁵ (MPI) es un estándar para una biblioteca de paso de mensajes, diseñado para funcionar en una amplia variedad de arquitecturas informáticas paralelas. MPI permite la comunicación entre procesos, mediante el envío y recepción de mensajes. Comúnmente se utiliza en sistemas de alto rendimiento²⁰ (HPC, por sus siglas en inglés) y entornos informáticos paralelos para desarrollar aplicaciones paralelas escalables y eficientes.

Al crear el entorno MPI en una aplicación, se ejecutan en paralelo varios procesos, cada uno con su correspondiente *id*, también llamado *rank*. El programador elige cuál va a ser el desempeño de los procesos. Por ejemplo, en el modelo *Master-Worker*, el primer proceso (*rank* = 0) generalmente, es llamado *master*, y se encarga de distribuir los datos entre los demás procesos, llamados *workers*. La Figura 2.1, muestra la comunicación entre los procesos en este modelo. El proceso *master* reparte los datos mientras que los *workers* los procesan y devuelven el resultado.

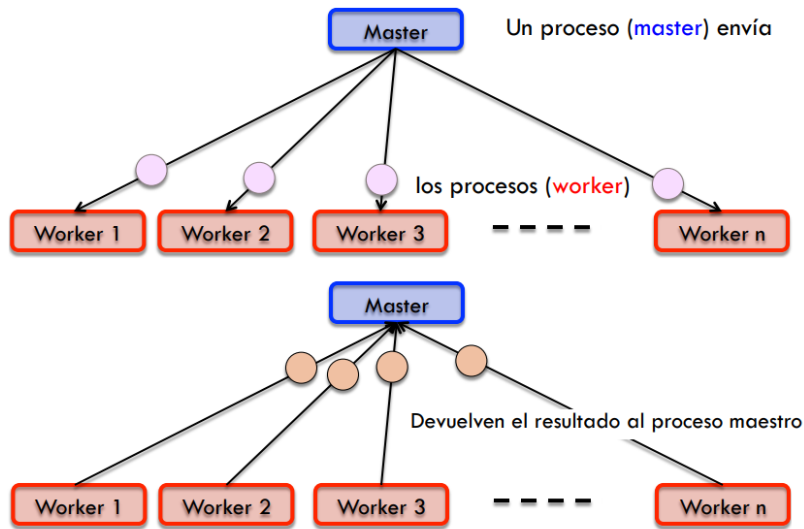


Figura 2.1: Comunicación Master-Worker

Esta técnica de paralelización utiliza memoria distribuida, es decir, cada proceso tiene su propia memoria local. Así, los procesos no tienen que preocuparse por los problemas de la memoria compartida, como la sincronización para el acceso de variables compartidas, condiciones de carrera o *deadlocks* (dos o más procesos quedan bloqueados en un estado en el que ninguno puede continuar con su ejecución, pues están esperando a que otro proceso, también bloqueado, libere un recurso necesario para continuar). Asimismo, la memoria compartida no es fácilmente escalable a un gran número de procesadores¹⁸.

Un programa ejecutado en paralelo, donde múltiples procesos se ejecutan en el mismo programa de manera independiente, pero trabajan con diferentes conjuntos de datos se denomina, por sus siglas en inglés, SPMD (Single Program Multiple Data). Este modelo es comúnmente utilizado en computación de alto rendimiento (HPC) y en entornos de procesamiento paralelo. La escalabilidad y eficiencia de este modelo son sus principales ventajas. Los mensajes pueden ser:

- Síncronos: El proceso receptor se queda bloqueado esperando el mensaje.
- Asíncrono: el receptor no se bloquea, por lo que puede adelantar código mientras espera a recibir el mensaje.

Un programa MPI (ver Figura 2.2), comparte el mismo código para todos los procesos ejecutados. Un proceso lee el conjunto de datos y los carga en su memoria, para luego dividirlos y enviarlos a los procesos disponibles. Una vez repartido el *dataset*, se ejecutan en paralelo y procesan los datos recibidos. Cuando un proceso finaliza el procesamiento de los datos, envía los datos procesados al proceso correspondiente. La Figura 2.3 representa en código esta idea.

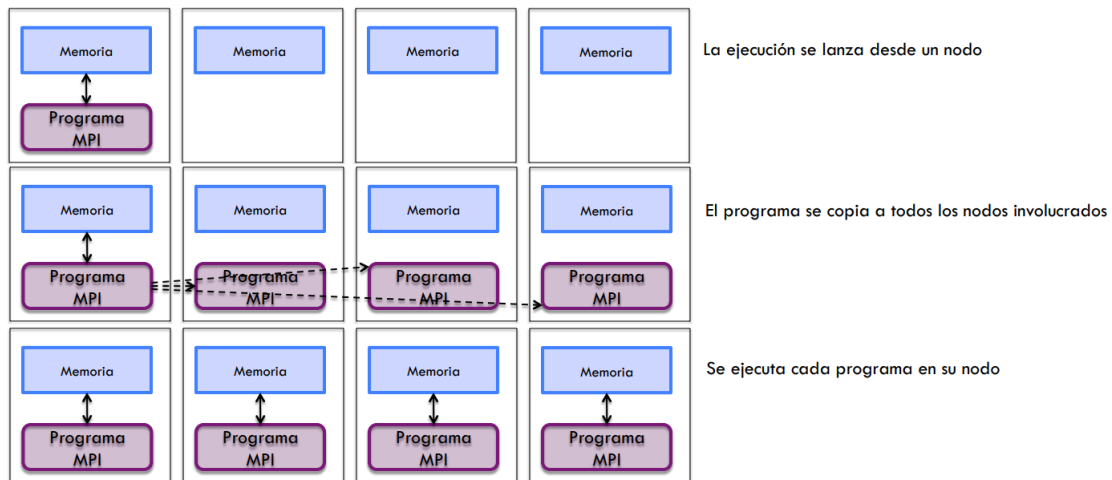


Figura 2.2: Ejecución MPI

```

1  from mpi4py import MPI # Al importar la biblioteca en Python se genera el
    entorno.
2
3  comm = MPI.COMM_WORLD      # Comunicador
4  status = MPI.Status()      # Status
5  myrank = comm.Get_rank()   # id de cada proceso
6  numProc = comm.Get_size()  # Numero de procesadores
7
8  if myrank==0:              # Master
9      # Carga el conjunto de datos. Los divide y envia.
10     # Recibe todos los datos procesados.
11 else:                       # Workers
12     # Recibe el subconjunto de datos que le asigna el Master.
13     # Procesa los datos. Los envia.

```

Figura 2.3: Esquema básico para ejecutar un programa MPI en Python

2.2. Aprendizaje por Refuerzo

Reinforcement Learning (RL, por sus siglas en inglés), en español, Aprendizaje por Refuerzo, es un tipo de aprendizaje automático donde el agente aprende en base a las decisiones tomadas al interactuar con el entorno. El agente aprende a cumplir un objetivo en un entorno ejecutando un determinado número de acciones. Este tipo de algoritmos no requiere de entradas etiquetadas como en el aprendizaje supervisado, sino que recibe una retroalimentación, *feedback* en inglés (recompensas o castigos), al realizar acciones en los estados. Aprendiendo con prueba y error, el agente explora el entorno para almacenar las mejores acciones para cada estado.

Los componentes esenciales del algoritmo son los siguientes:

- *Agente* que interactúa con el entorno y aprende de él, ejecutando sus acciones.
- *Entorno* con el cual el agente interactúa. Responde a las acciones tomadas por el agente y provee el *feedback*.
- Conjunto de acciones o *decisiones* que el agente puede realizar.
- *Estados*, son las configuraciones que el entorno puede tomar.
- *Feedback*, recompensas o castigos del entorno al realizar una acción en un estado.
- Condición de *finalización*. La cual puede ser desde encontrar la función óptima, hasta realizar un número de acciones.

2.2.1. Algoritmo Q-Learning

El algoritmo Q-Learning es una mezcla entre programación dinámica y Monte Carlo²². Es el más básico de entre los algoritmos de aprendizaje por refuerzo. Se usa para encontrar la mejor política de selección de acciones para un proceso de Decisión de Markov Determinado (MDP, por sus siglas en inglés)⁸.

El procedimiento se realiza actualizando iterativamente las estimaciones de calidad de realizar dicha acción en el estado actual, conocido como valor-Q. Se suele representar en forma de matriz $Q(S,A)$, guardando los valores-Q de las acciones en los estados. Este valor representa cómo de buena es la acción a realizar en un estado después de realizar una etapa de entrenamiento. Para ello, la Figura 2.4 muestra la fórmula para actualizar los valores, para cada acción tomada por el agente.

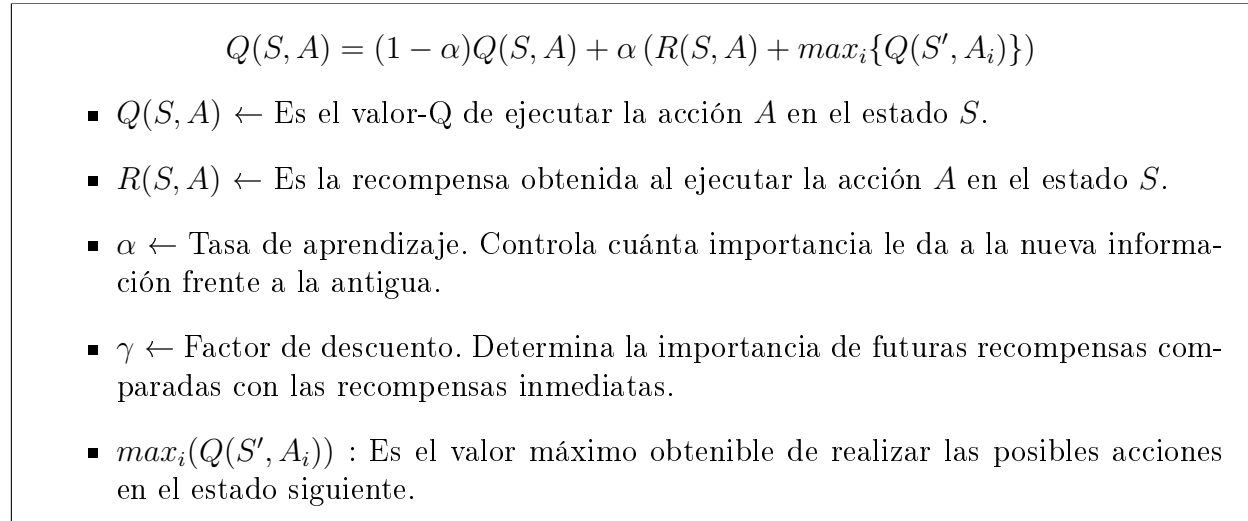


Figura 2.4: Cálculo del Q-Value de un estado y acción

El agente toma la decisión de ejecutar una acción dependiendo del hiper-parámetro ϵ con valores entre $[0,1]$. Con un número aleatorio (en el mismo intervalo) calcula la probabilidad de ejecutar la mejor acción aprendida hasta el momento, o una acción aleatoria entre las disponibles. Si el valor es alto, con alta probabilidad se ejecutará la mejor acción aprendida hasta el momento, y es posible que no aprenda otras formas de alcanzar el objetivo.

Este algoritmo se ha aplicado en muchos dominios, como puede ser videojuegos de Atari¹⁴, robótica o problemas de optimización. Sin embargo, sufre cuando el entorno tiene muchos estados, ya que la complejidad espacial aumenta considerablemente, y no resulta práctico contar con dos matrices. Por eso se diseñó el algoritmo DQN, el cual usa una red neuronal. Así, se elimina la maldición de dimensionalidad¹¹, problemas que surgen con el ex-

ceso de variables independientes en un *dataset*. En este algoritmo, el problema es el elevado número de estados que el agente ha de recorrer.

2.2.2. Deep Q-Network (DQN)

Debido a los problemas de escalabilidad mencionados anteriormente, se desarrolló el algoritmo de Redes Neuronales Profundas (DQN, por sus siglas en inglés). Este algoritmo combina redes neuronales con la base de aprendizaje por refuerzo, eliminando así la Q-Table.

La estructura de la red neuronal depende del entorno del problema. Los valores de la capa oculta se pueden modificar dependiendo de las necesidades del programador, pero la capa de entrada y salida depende del problema. La entrada se adapta para recibir un estado del entorno, como por ejemplo una imagen representada como una matriz. La salida de la red tendrá tantos nodos como acciones tenga el agente.

En este trabajo, el entorno del problema será el juego Pacman, diseñado por la empresa *Namco*, y en particular la versión de *Atari 2600*, (ver Figura 2.5). El juego consiste en recolectar todas las monedas del laberinto sin ser comido por un fantasma. Implementamos el juego -desde cero- para moldear según nuestros intereses la implementación y que el algoritmo DQN sea más eficiente y sencillo. En el Capítulo 3, diseño e implementaciones, se desarrolla en profundidad.

El algoritmo DQN a realizar tendrá dos redes neuronales, una del estado actual y otra de anticipo, es decir, el siguiente estado. Esto sirve para ayudar a tener más contexto del estado actual, pues con una sola imagen (estado del juego), la información del estado puede variar considerablemente. En la fase de entrenamiento se realizan varios episodios, que consisten en ejecuciones hasta que se dé una condición de finalización. Además de ejecutar repeticiones de estados guardados anteriormente (replay buffer). En este algoritmo se usan tres hiperparámetros:

1. *Gamma*, factor de descuento $[0, 1]$. Utilizado para saber cuánto resta a la recompensa adquirida al realizar una acción en un estado.

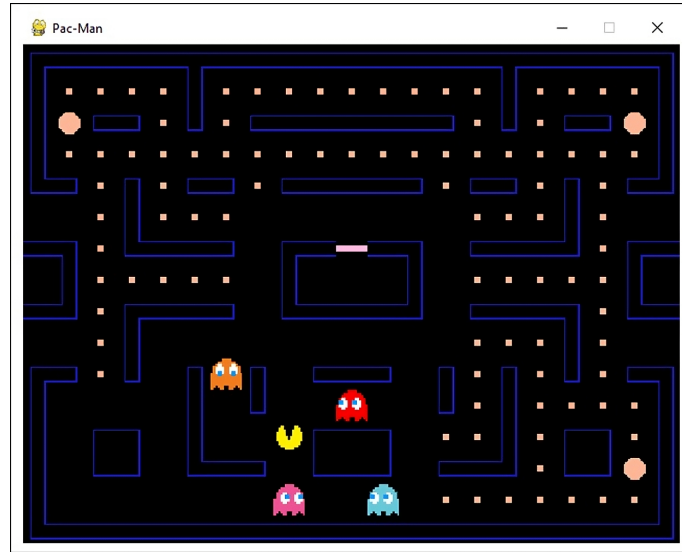


Figura 2.5: Juego Pac-Man implementado desde cero. Versión Atari 2600

2. *Epsilon*, tasa de exploración $[0, 1]$. Probabilidad utilizada para ejecutar una acción aleatoria o la mejor hasta el momento.
3. *Learning rate*, tasa de aprendizaje $[0, 1]$. Para la propagación hacia atrás de las redes neuronales. Esencialmente, mide cuánto cambian los pesos de los nodos al tener un fallo.

Además de estos parámetros, el algoritmo cuenta con otras variables para desarrollar las redes neuronales.

- *Epsilon decay*. Utilizado para no usar siempre el mismo valor de *epsilon*. Esta variable marca cuanto se reduce la variable *epsilon* entre episodios.
- Número de ejemplos de entrenamiento (*batch size*). Se utiliza para actualizar los parámetros de la red neuronal durante una sola iteración del entrenamiento.
- Tamaño de la capa oculta. Marca el número de neuronas en cada capa.

2.3. Aprendizaje No-Supervisado

Los métodos no supervisados (unsupervised methods, en inglés) son algoritmos de aprendizaje automático que basan su proceso en un entrenamiento con datos sin etiquetar. Es decir, a priori, no se conoce ningún valor objetivo, ya sea categórico o numérico. La meta de este aprendizaje es encontrar patrones o estructuras en los datos proporcionados. Estos algoritmos son útiles en escenarios en los cuales hay escasez de datos etiquetados o éstos no están disponibles.

Hay muchos tipos de técnicas de aprendizaje no supervisado como, entre otros, la detección de anomalías, reducción de dimensionalidad o *clustering*. En este proyecto vamos a reducir el tiempo de ejecución de las técnicas de *clustering* que se encargan de agrupar individuos basándose en alguna medida de similitud. Como no es aprendizaje supervisado, no disponemos de información categorizada previamente, por lo que hay que calcular el número óptimo de *clusters*. Para ello, hay medidas ya estudiadas como el diagrama de “codo”, cuyo valor óptimo de *clusters* se calcula visualmente, cuando empieza a crearse un codo (la diferencia con el número anterior no es tan pronunciada como en puntos anteriores). Hay otros coeficientes que calculan la optimalidad con algoritmos, como el coeficiente de Davies-Bouldin, cuyo valor mínimo indica el número óptimo de *clusters*, o el coeficiente de Silhouette, similar al anterior, pero con el valor máximo. Se pueden apreciar los diferentes coeficientes para una misma categorización en la Figura 2.6. Como se puede apreciar, el diagrama de codo es el coeficiente más complicado de visualizar, lo otros dos coeficientes solo es necesario encontrar el menor o mayor valor, mientras que en el primero hay que visualizar el codo, y en este ejemplo se podría elegir también tres *clusters* como número óptimo.

Los llamados métodos jerárquicos³ tienen por objetivo agrupar *clusters* para formar uno nuevo o bien separar alguno ya existente para dar origen a otros dos, de tal forma que, si sucesivamente se va efectuando este proceso de aglomeración, se minimice alguna distancia o bien se maximice alguna medida de similitud.

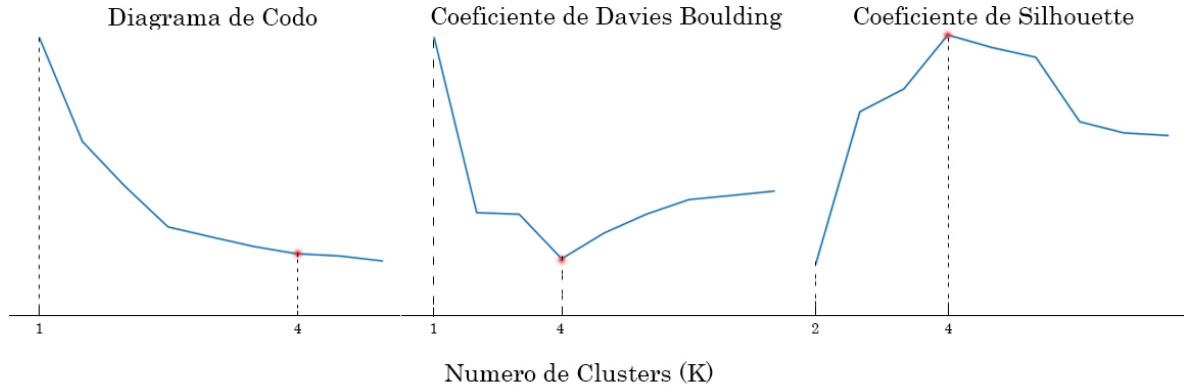


Figura 2.6: Coeficientes de agrupación

2.3.1. Clustering jerárquico aglomerativo

Este algoritmo usa una matriz para realizar la agrupación de los individuos. Comienza teniendo N *clusters*, uno por cada individuo de la población. La matriz se representa por las filas, es decir, la fila i -ésima representa el *cluster* i -ésimo. La matriz se rellena con las distancias entre los *clusters*, por lo que la celda (i, j) representa la distancia entre el *cluster* i y el j .

En cada iteración, se busca en la matriz la celda (i, j) con menor valor (distancia mínima), y se juntan los *clusters* que representan la fila i con la columna j . La matriz se actualiza, eliminando la fila y la columna con mayor índice (entre i, j), y actualizando la fila y columna de menor índice. Este proceso se repite hasta que solo haya un *cluster*. Las distancias entre *clusters* pueden ser:

- Centroides: cada *cluster* tiene un centro.
- Enlace simple o compuesto: la distancia entre *clusters* viene dada por la menor o mayor distancia, respectivamente, entre los individuos que representan cada *cluster*.

La complejidad en los enlaces simple y completo tienen un coste cúbico $O(N^3)$, al tener que comparar todos los individuos uno a uno entre dos *clusters*.

Al finalizar la ejecución se puede representar la agrupación mediante un dendograma⁷ (ver Figura 2.7), y comprobar el número óptimo de *clusters* para la población calculada. Sin embargo, no es igual de preciso que los coeficientes mencionados anteriormente.

Algorithm 1: Jerárquico Aglomerativo

```
Data: poblacion, C // Numero de clusters deseados
Result: agrupacion // Clusters para cada individuo de la poblacion
D := init() // Inicializar la matriz de distancias
while number of rows in matrix > C do
    // Recorre la matriz en búsqueda del menor valor (i, j)
    i, j :=busqueda_min(poblacion)
    // Agrupa los cluster (i, j)
    agrupacion := agrupar_clusters(poblacion, i, j)
    // Elimina la fila y columna de mayor índice
    eliminar_cluster(poblacion, max(i, j))
    // Calcula nuevas distancias al cluster agrupado (i)
    nuevas_distancias(poblacion, min(i, j))
```

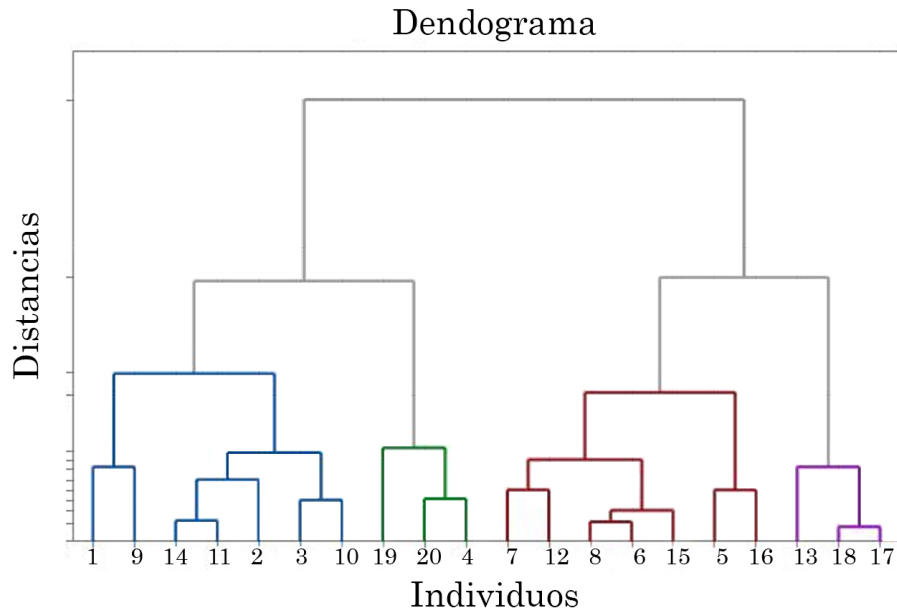


Figura 2.7: Dendograma de una población de 20 individuos

2.3.2. Clustering basado en particiones: K-Medias

La meta de este algoritmo es particionar la población inicial en K *clusters*, donde cada individuo se agrupa con el *cluster* más próximo. El algoritmo itera en un bucle en el cual calcula la asignación de todos los individuos. Al final de cada iteración calcula los nuevos centros, si estos no cambian con respecto a la iteración anterior, termina la ejecución. La

optimalidad de la agrupación se calcula mediante el sumatorio de las distancias de todos los individuos y los centroides de sus *clusters*.

Algorithm 2: K-Medias

Data: poblacion, K

Result: agrupacion

centrosNuevos := init(K) // Inicializa los centros de manera aleatoria

repeat

 centros := centrosNuevos

 // Asigna a cada individuo el cluster mas cercano

 agrupacion := asignar(poblacion, centros)

 // Calcula las posiciones de los nuevos centros

 centrosNuevos := calculaCentros(poblacion, asignacion)

until *centros* != *centrosNuevos*;

return *agrupacion*;

Sin embargo, hay que tener en cuenta que la inicialización de los centros es estocástica, por lo que el algoritmo puede converger en un óptimo local. Por eso es importante repetir el algoritmo varias veces para encontrar el óptimo general. La Figura 2.8 muestra un ejemplo de ejecución, junto con la búsqueda de la mejor agrupación. Esta figura representa la búsqueda al ejecutar tres veces el algoritmo. El intento con menor valor del sumatorio de distancias de individuos, y su *cluster* asignado, es la mejor asignación entre los intentos ejecutados. El tercer intento es la agrupación que muestra la figura.

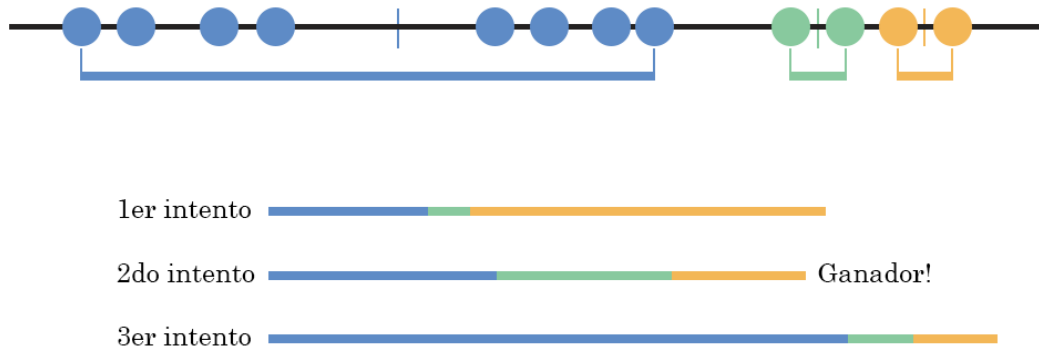


Figura 2.8: Búsqueda del óptimo general en el algoritmo de K-Medias

2.4. Aprendizaje Supervisado

Al contrario que el apartado anterior, este tipo de aprendizaje automático es entrenado con un *dataset* categorizado con su salida correcta. El algoritmo aprende de este conjunto para hacer predicciones sobre unos datos desconocidos.

El objetivo de este algoritmo es aprender la función que mapea las variables de entrada en las categorías correctas de salida. Para ello, ajusta los parámetros con técnicas de optimización iterativas para minimizar el error en sus predicciones.

Los ejemplos más comunes son la clasificación, para dividir la población en categorías según unos parámetros, y regresión, que encuentra las correlaciones entre las variables dependientes e independientes.

2.4.1. K-Vecinos más Cercanos - KNN

KNN (K-Nearest Neighbors en inglés) es un algoritmo simple pero potente, resultando muy efectivo para tareas de clasificación y regresión. Se basa en la idea de que los puntos de datos similares tienden a agruparse en el espacio de características. Este algoritmo pertenece al paradigma de aprendizaje perezoso o basado en instancias.

- Perezoso: no calcula ningún modelo y demora todos los cálculos hasta el momento en que se le presenta un ejemplo nuevo.
- Basado en instancias: usa todos los individuos disponibles y ante un ejemplo nuevo recupera los más relevantes para componer la solución.

No hay una forma de determinar el mejor valor para K, de forma que hay que probar con varias ejecuciones. Valores pequeños de K crea ruido, provocando que inicialmente se categorice con un *cluster* no idóneo, y a la larga se categoricen muchos de forma incorrecta. Valores grandes con pocos datos favorece a los *clusters* con más individuos. Un valor diferente de K puede cambiar la categoría de un individuo. En la imagen izquierda de la Figura 2.9 se puede apreciar el proceso de asignación de un *cluster* a un nuevo individuo. Como se puede

ver con los círculos que delimitan los K vecinos más cercanos, si variamos K, la asignación del nuevo individuo puede cambiar.

Algorithm 3: K-Vecinos más Cercanos

Data: poblacion, etiquetas, poblacionPred, K
Result: agrupacion // Clusters para cada individuo de la población
agrupacion := $\emptyset$
for *each* individuo ind *in* poblacionPred **do**
 // Recorre toda la población guardando los K individuos más cercanos
 vecinos := recorrer_poblacion(poblacion, individuo, K)
 // Clasifica según los K vecinos
 cluster := clasificar_individuo(vecinos)
 agrupacion.append(cluster)
return agrupacion

La distancia entre individuos más usada es la Euclídea, pero requiere más tiempo al aplicar potencias y raíces cuadradas en su cálculo. La distancia Manhattan es más rápida, pero menos precisa.

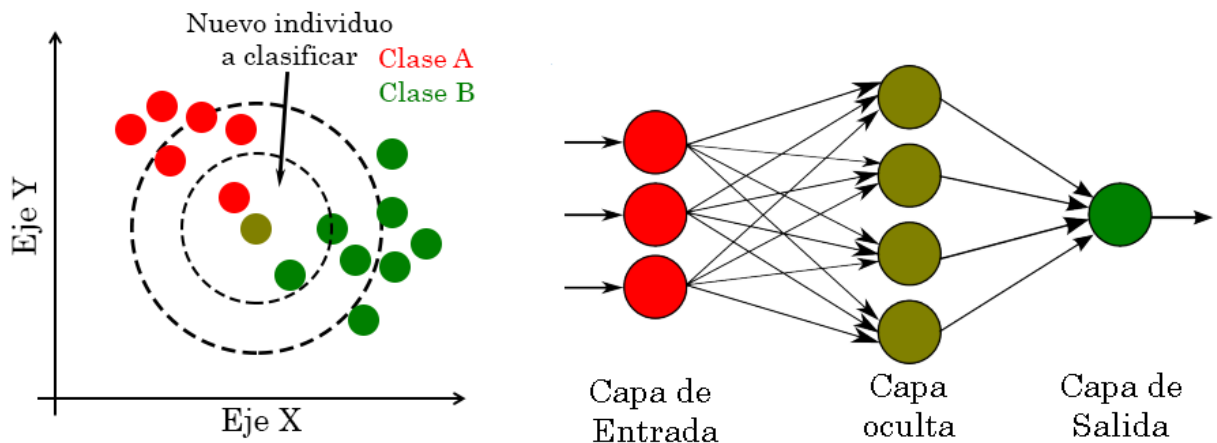


Figura 2.9: Algoritmos de aprendizaje supervisado

2.4.2. Redes Neuronales

Las redes neuronales son un modelo computacional inspirado en el funcionamiento y estructura de las neuronas del cerebro humano. Esencialmente, consisten en capas de nodos

interconectados, llamadas neuronas artificiales. La estructura del modelo se muestra en la imagen de la derecha de la Figura 2.9. En este modelo se aprecian:

- Capa de entrada, en la cual, habrá tantas neuronas como variables de entrada tenga el modelo de predicción.
- Capa oculta, representada con una o más capas internas. Cada una contiene un número determinado de neuronas.
- Capa de salida. Como en la entrada, tendrá un número de neuronas relacionadas con las variables de salida.

Se ha demostrado que las redes neuronales tienen un rendimiento notable en muchas tareas, como el reconocimiento de imágenes o procesamiento de lenguaje natural. Aprenden patrones complejos al someterse a un entrenamiento específico con un amplio *dataset* categorizado.

En el proceso de entrenamiento aprende a realizar una tarea específica ajustando los parámetros internos (pesos en las conexiones), gracias al *dataset* proporcionado. Normalmente, este ajuste se lleva a cabo con algoritmos de optimización como descenso de gradiente, donde se comparan las predicciones del modelo con la categoría correcta, y se actualizan los parámetros del modelo con un método de propagación hacia atrás, *backpropagation* en inglés. Estos valores se actualizan dependiendo del error cometido y la tasa de aprendizaje proporcionada al modelo.

El Algoritmo 4 muestra la etapa de entrenamiento a la cual una red neuronal se somete para poder cambiar los pesos de las neuronas y poder obtener un funcionamiento correcto con respecto a los valores entrenados.

Algorithm 4: Red Neuronal

```
Data: entrenamiento, etiquetas, evaluacion // Individuos sin categorizar
      repeticiones, capas // Tam. entrada, oculta, salida
Result: pesos // Opcionalmente, devolver los pesos de la red
pesos := init() // Inicializar los pesos de manera aleatoria
for rep ← 0 to repeticiones do
  cont := 0
  for each ind in entrenamiento do
    // Suma el valor recibido de la capa anterior multiplicada por los pesos de la
    // capa actual con la siguiente. Así se determina la importancia de conexión
    // entre las neuronas. Con el valor calculado se aplica a una función de
    // activación y se pasa a la siguiente capa hasta llegar a la salida.
    prediccion := forward(pesos, ind)
    // El valor predicho calculado en la salida es comparado con la etiqueta, y se
    // calcula el error. Este error se manda para atrás actualizando los pesos. Se
    // suma la multiplicación del valor predicho en cada capa con la tasa de
    // aprendizaje y el error.
    backpropagation(pesos, prediccion, etiqueta[cont]) cont++
return agrupacion
```

2.5. Algoritmos Evolutivos

La programación evolutiva es una técnica de optimización inspirada en la teoría de la evolución biológica. Se basa en el concepto de selección natural y evolución de las poblaciones para encontrar soluciones a problemas complejos.

La población está compuesta por individuos, que pueden ser representados con arrays de números reales, binarios o un árbol. Los individuos tienen un cromosoma, que a su vez tiene uno o varios genes, con uno o más alelos. Esta población es sometida a métodos de evaluación, selección, cruce y mutación para, con el paso de las generaciones, maximizar o minimizar un valor *fitness*.

Esta técnica es muy útil para problemas de optimización donde los métodos tradicionales no proporcionan el rendimiento deseado. Los Algoritmos Evolutivos se han aplicado a varios dominios, como por ejemplo la bioinformática o robótica⁶.

La Figura 2.10 muestra el diagrama de estados del algoritmo más básico. Cada método se

puede modificar para cualquier individuo, así como añadir más técnicas para garantizar y/o mejorar los resultados finales, como puede ser el elitismo, que garantiza la supervivencia de los mejores individuos, o un desplazamiento de los valores *fitness* para evitar valores negativos.

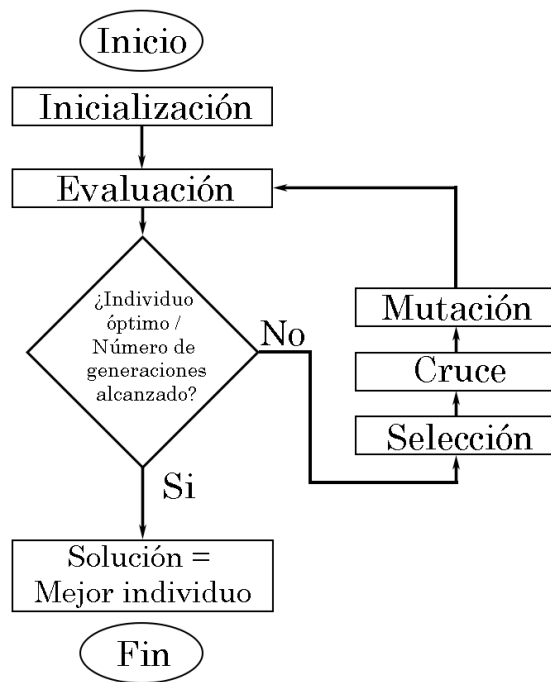


Figura 2.10: Algoritmo Evolutivo básico

Capítulo 3

Diseño e Implementación de estrategias para aumentar el rendimiento de algoritmos de IA

En este capítulo se presentan los diseños e implementaciones desarrollados a lo largo del trabajo. Inicialmente se presenta una introducción a la biblioteca MPI, con varios ejemplos donde se mejora el rendimiento de programas sencillos fuera del ámbito de la inteligencia artificial. Posteriormente, se describen los algoritmos de IA ordenados -de menor a mayor- según su complejidad.

3.1. Programas sencillos

Para introducir MPI en el proyecto se implementan -utilizando esta biblioteca- varios programas sencillos. Primero, la multiplicación de matrices, que tiene un coste cúbico $O(N^3)$, al tener que recorrer, para cada elemento de la matriz, una fila y columna entera. Segundo, algoritmos de ordenación que, para simplificar, solo se realiza un estudio de las ordenaciones con mayor complejidad temporal, $O(N^2)$ y MergeSort con coste $O(N * \log N)$.

Las matrices son un concepto matemático muy relevante en el mundo de los videojuegos y en el ámbito de la inteligencia artificial. Hay muchas técnicas de IA que conllevan la gestión de imágenes -representadas digitalmente como matrices de píxeles- como en el algoritmo DQN cuya red neuronal tiene como entrada varios fotogramas para poder aprender

La multiplicación de matrices es un buen ejemplo para presentar una estrategia basada en MPI, debido su alto coste computacional. Para ello, hay que plantear cómo dividir el trabajo entre los procesos. Inicialmente, se puede pensar que es mejor enviar los datos conforme se finaliza una operación, pero en esta operación se necesitan las filas de una matriz y columnas de otra, por lo que conviene que cada proceso tenga una matriz entera en su memoria local para agilizar el proceso y poder enviar más datos al mismo tiempo. Cada *worker* se va a encargar de un determinado número de filas, paralelizando así el cálculo. El *master* se encarga de dividir la matriz entre los procesos. Se puede abordar con dos enfoques distintos:

- Reparto estático: los datos se asignan antes de empezar el cálculo.
- Reparto dinámico: los datos se reparten en tiempo de ejecución, asignando los mismos de forma proporcional a la velocidad de cada proceso.

En la primera estrategia, dividimos la segunda matriz entre todos los *workers*, dejando al *master* en espera de recibir datos. Esta mejora depende de la velocidad de los procesos, pues se puede generar un cuello de botella si todos terminan y envían los datos al mismo tiempo. Además, se aumenta la complejidad espacial entre los procesos, pues se divide la matriz entera entre los *workers*.

En la segunda estrategia, hay un flujo constante de nuevos datos y resultados obtenidos, reduciendo el tiempo perdido en un posible cuello de botella. En la Figura 3.1 se muestra como el *master* divide la matriz, marcando en negro la parte ya procesada. A su vez cada *worker* tiene una sola fila en su memoria, reduciendo la complejidad espacial.

Los algoritmos de ordenación tienen que iterar varias veces hasta que el array de elementos esté completamente ordenado. Los métodos pueden variar considerablemente el tiempo de ejecución.

Para las ordenaciones cuadráticas, los métodos populares como *BubbleSort*, *InsertionSort* y *SelectionSort*, han sido estudiados y optimizados para que, aunque tengan un coste cuadrático $O(N^2)$, en el caso peor, proporcionen un buen rendimiento. Basándose en estos

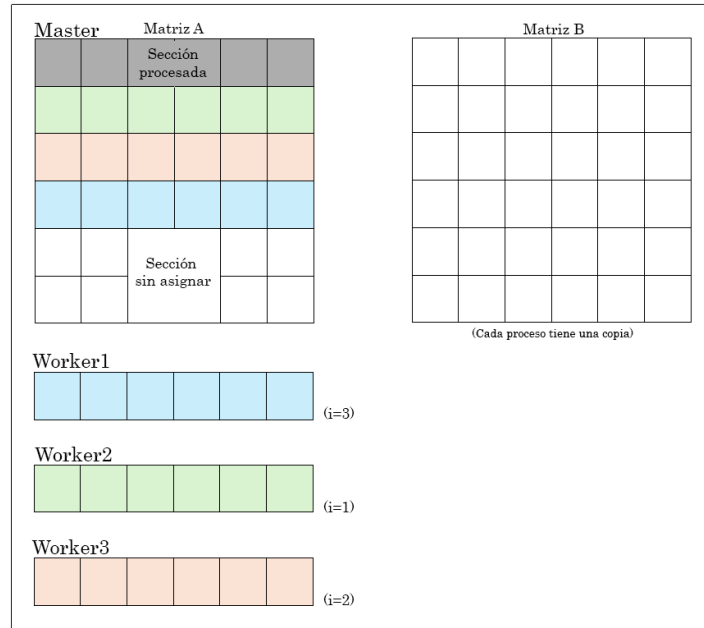


Figura 3.1: División de datos entre los procesos *workers* en la multiplicación de matrices

algoritmos, se ha diseñado uno adicional llamado *SequentialSort*. Este algoritmo recorre todas las posiciones del array y, para cada elemento, compara todos los datos, sumando en un contador los elementos mayores que él, para calcular así su posición en el array ordenado. Una vez finalizada una iteración, se coloca el elemento en el array ordenado, si la posición actual está ocupada es porque hay una repetición del elemento, y se tiene que colocar en la siguiente celda libre. La Figura 3.2 representa el proceso de ordenación, marcando en gris el elemento que ha de compararse con los demás en la iteración i -ésima.

Este método siempre tendrá coste cuadrático $O(N^2)$. No es como los anteriores que van reduciendo el espacio conforme aumentan las iteraciones, pero es fácilmente paralelizable.

Para lograr reducir el tiempo de ejecución para esta ordenación cuadrática, se desarrollan las dos estrategias siguientes, cada cual con sus ventajas e inconvenientes.

1. Enviar a todos los *workers* el array entero para que trabajen de manera independiente.
2. Dividir el array entre los *workers* para trabajar conjuntamente.

En la primera estrategia el *master* envía a todos los *workers* el array entero. Una vez

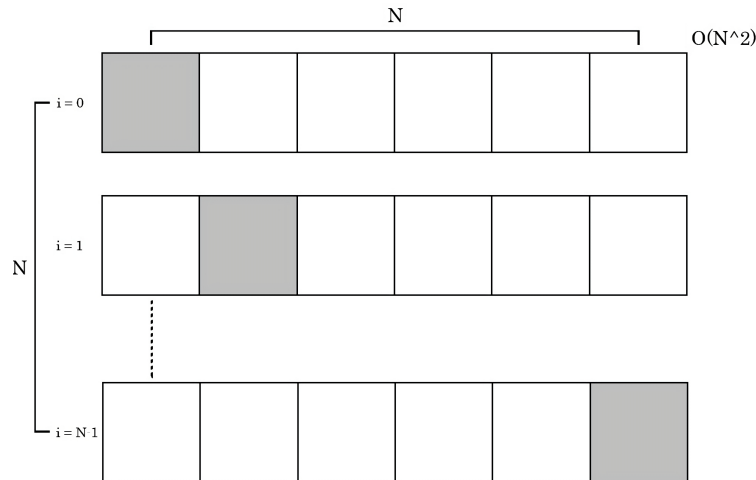


Figura 3.2: Iteraciones del algoritmo *SequentialSort*

recibido el array de elementos, el *master* envía elementos sin procesar del array original a los *workers* disponibles. Al recibir un elemento lo procesan (hacen las comparaciones), y envían la posición del elemento al *master*, recibiendo de vuelta otro si todavía faltan elementos por procesar.

No obstante, la segunda estrategia logra reducir el uso de memoria de tal forma que entre todos los procesos ejecutados solo haya dos copias del array que hay que ordenar, en lugar de mantener M copias (siendo M el número de procesos ejecutados). En cada iteración, el *master* envía un elemento a todos los *workers*. Estos hacen todas las comparaciones en sus subarrays y devuelven cuantos elementos pertenecientes a su subarray son mayores que el recibido. Ambas estrategias tienen la misma complejidad temporal.

Los algoritmos de ordenación logarítmicos son muy útiles y eficientes. *QuickSort* tiene varios problemas como la profundidad de recursión y en el caso peor es cuadrático. Los algoritmos de *RadixSort* y *HeapSort* son eficientes sin aplicar mejoras, y *MergeSort* es muy popular, tanto que se aplica en *TimSort*⁴ método de ordenación por defecto en Python. Este último combina *InsertionSort*, una ordenación cuadrática muy eficiente para ordenar pequeños conjuntos de datos, teniendo una baja sobrecarga en términos de operaciones, para luego usar las mitades ordenadas con *MergeSort*. Sin embargo, el algoritmo básico de

MergeSort no es tan eficiente.

Aplicando la misma idea que *TimSort*, se puede mejorar el tiempo de ejecución de *MergeSort*, aplicando combinaciones de los métodos básicos con complejidad cuadrática y comprobar la eficiencia. Esta estrategia consiste en crear varios procesos (para mayor eficacia y simplicidad, el número de procesos tiene que ser potencia de dos), y se divide el array entre los procesos. Las fases de esta estrategia son:

- Primera fase de ordenación: cada proceso ordena su subarray con el método de ordenación correspondiente. En el Capítulo 4, se realiza un estudio de los algoritmos cuadráticos, donde SelectionSort es el algoritmo que mejores resultados obtiene.
- Segunda fase de reagrupación y ordenación: esta fase se repite hasta solo tener un proceso activo, es decir, el array esté completamente ordenado.

En la comunicación entre procesos, cada uno se conecta con el proceso activo más cercano (según su *rank*). El proceso de mayor *id* (*rank*) envía su array ordenado y finaliza su ejecución. El proceso receptor se encarga de ordenar ambas mitades en una sola. Utilizando una barrera llamada `MPI_Barrier`, garantizamos que todos terminen al mismo tiempo. Esta mejora aplica la idea de sincronización con *barrera simétrica mariposa*, técnica de sincronización que conecta los procesos dos a dos, aumentando la distancia de los procesos para que, en aproximadamente K iteraciones ($2^K = M = \text{número de procesos}$), todos los procesos estén sincronizados. Para la estrategia implementada, se sincronizan por orden de cercanía entre *ids*. La Figura 3.3 muestra el proceso de sincronización. En cada iteración se finaliza la ejecución de los procesos en rojo, así hasta tener un único proceso con el array entero ordenado.

Para aplicar MPI y paralelizar programas hay que tener en cuenta que la comunicación entre procesos requiere un tiempo para enviar/recibir mensajes. Si queremos reducir el tiempo de ejecución de un programa tenemos que asegurarnos que la estrategia es viable para mejorar el rendimiento. Si ejecutamos, por ejemplo, una búsqueda lineal en un array, a

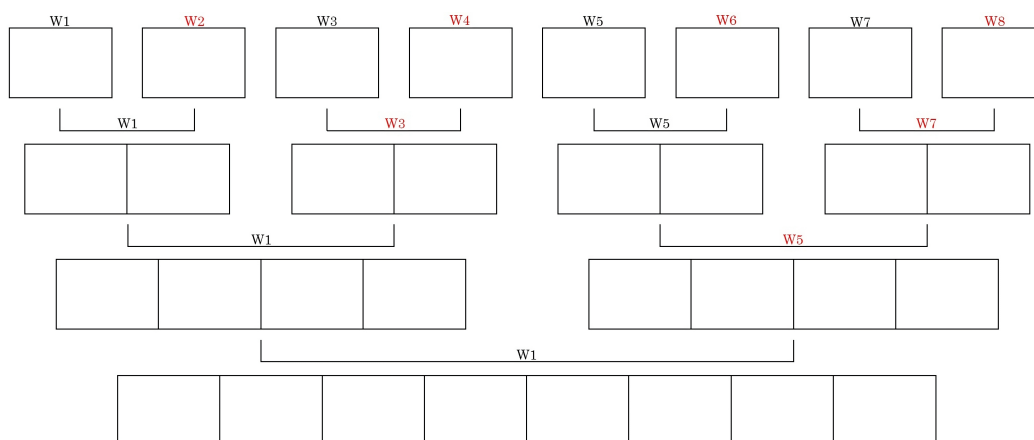


Figura 3.3: Ejecución de *MergeSort* con 8 procesos *worker* $W_1 \dots W_8$

primera vista, reducir el espacio de búsqueda puede ser beneficioso. Dividiendo el espacio de búsqueda entre los *workers* reduce el tiempo de $O(N)$ a $O(N/\text{numWorkers})$. Pero ¿se puede reducir el tiempo de ejecución al dividir el espacio entre los *workers*?

Para responder esta pregunta es necesario tener en cuenta el tiempo de paso de mensajes (overhead). Si no se tuviese en cuenta, se podría garantizar la reducción, pero la comunicación entre procesos tiene un coste, y con un tiempo lineal, generalmente no se pueden lograr mejoras, más bien aumenta el tiempo de búsqueda.

Por este motivo, hay que tener en cuenta la complejidad temporal de los algoritmos que queremos optimizar, ya que no siempre es eficiente aplicar paralelismo.

3.2. Algoritmos de Clustering

Una vez introducido MPI con programas básicos, podemos presentar las implementaciones de los algoritmos relacionados con la inteligencia artificial. Las técnicas de clustering toman una población y, dependiendo del conjunto de datos, categorizan los individuos. Los algoritmos pueden ser supervisados, si además de la población a categorizar, tenemos una población categorizada previamente, o no-supervisados, si no contamos con esta población etiquetada.

3.2.1. Jerárquico Aglomerativo

Este algoritmo de aprendizaje no supervisado usa una matriz para calcular las agrupaciones. Como es una matriz simétrica, podemos reducir la complejidad espacial usando solo el triángulo superior.

La distancia entre *clusters* es muy importante. Además de calcular agrupaciones distintas, también varía la complejidad temporal. La más eficaz y rápida es la de centroides, para calcular la distancia entre dos *cluster* solo necesita el cálculo entre dos puntos (los centros de los *clusters*). El cálculo de la distancia por enlace simple y completo, es más complejo. Cada *cluster* almacena las coordenadas de sus individuos, para, a la hora de calcular la distancia entre dos *clusters* (C_i y C_j), comprobar la distancia de cada par de puntos (donde uno pertenece a C_i y el otro a C_j). La nueva distancia usando enlace simple es la mínima distancia entre cualquier par de puntos, mientras que la completa es la máxima.

Una vez comentadas las estrategias en el cálculo de multiplicación de matrices, podemos usar estas para mejorar este algoritmo. La primera idea de enviar las filas conforme se realizan los cálculos no se puede aplicar. El algoritmo es más complejo que realizar sumatorios de multiplicaciones (suma de productos, para la multiplicación de matrices), pues la matriz está en constante cambio. Tendría que realizarse un proceso de comunicación constante para gestionar la matriz. Esto y añadir más operaciones del algoritmo para agrupar los individuos, provoca que no sea viable realizar esta mejora.

La estrategia que se va a implementar consiste en dividir la matriz entre los *workers*. Cada proceso se encarga de una zona, paralelizando así el trabajo a realizar. Como es una matriz simétrica y se representa con el triángulo superior, hay que dividir la carga de trabajo equitativamente. No podemos implementar una mejora sin dividir el espacio de forma equitativa entre los *workers*. Si dividimos las filas de forma secuencial, el primer *worker* tendrá muchos más elementos que el último, parando la ejecución por “culpa” del primer proceso. La Figura 3.4 muestra el cálculo de elementos a procesar entre el primer y último *worker* si no se divide el espacio de manera equitativa.

$$\sum_{i=1}^{\text{filas}} (N - i) \gg \sum_{i=\text{filas}(M-1)}^{\text{filas}(M-1)+\text{filas}} (N - i)$$

N individuos de la población, M procesadores. N/M filas para cada *worker*.

Con 100 individuos de población y 4 *workers*, cada uno tendrá 25 filas. Por lo que:

- W_1 tiene las filas de 1-25, con 2175 elementos.
- W_4 , las filas de 76-100, con solo 300 elementos.

W_1 tiene 7.25 veces más elementos, no se reducirá el tiempo de ejecución.

Figura 3.4: Cálculo del número de elementos a procesar usando una distribución secuencial de filas

Dividiendo las filas por pares (parte superior e inferior) conseguimos una distribución mucho más eficiente. La Figura 3.5 muestra cómo se distribuyen las filas en cada *worker*. Así, cada *worker* tiene aproximadamente el mismo número de elementos que calcular y analizar, inicialmente. Sin embargo, puede variar si el número de filas no es divisible entre el número de procesos *workers*.

Una vez descrito el reparto del espacio entre los procesos, cada uno tiene que ejecutar el algoritmo en paralelo, sincronizándose cada cierto tiempo para actualizar valores. Refrescando la memoria, este algoritmo, en cada iteración, agrupa los dos individuos más cercanos, eliminando una fila y columna de la matriz. El bucle principal de la estrategia se repite hasta que solo haya C clusters. Las etapas son las siguientes:

1. El *master* pide a los *workers* la celda con menor valor (menor distancia entre los *clusters* i y j , siendo estos la fila y columna).
2. El *master*, con los valores recibidos, pide la fila (i) y la columna (j) del *worker* con menor distancia. Con estos datos, el *master* envía a todos los procesos estos índices, así como los *ids* de los *workers* que tienen que eliminar o actualizar la fila con mayor o menor índice respectivamente.

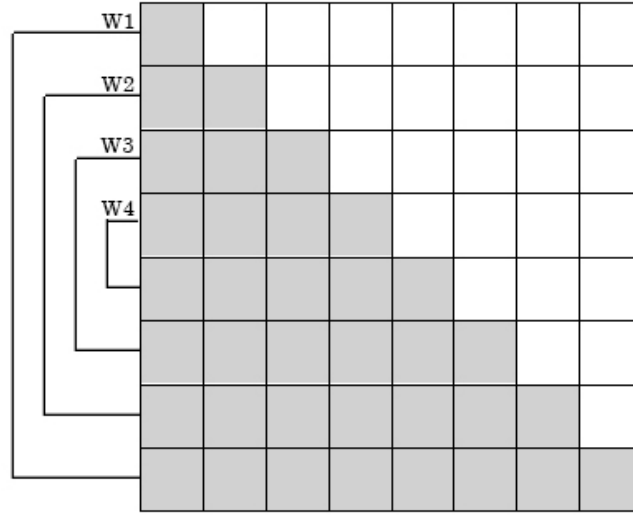


Figura 3.5: División de las filas utilizada en la estrategia desarrollada para el algoritmo Jerárquico Aglomerativo

3. Todos los *workers* eliminan la columna con mayor índice. El *worker* que tiene que eliminar la fila la elimina. Mientras que el que tiene que actualizar la actualiza, con o sin ayuda de los demás *workers*.

El cálculo de las distancias de la nueva fila (actualización) usando la distancia centroide es lineal, no se puede reducir el tiempo de ejecución. Pero para los enlaces simples y completos, que tienen un coste cuadrático, se debe intentar reducir el tiempo de cómputo. Para lograrlo se generan otras dos estrategias más, que lo único que cambian de esta primera estrategia es cómo se divide el cálculo de la actualización de la fila entre todos los *workers*.

El objetivo es encontrar una manera lo más optimizada posible de realizar el cálculo.

- Cuando hay pocos individuos por *cluster*, es más probable que haya muchas columnas que actualizar, y conviene dividir el cálculo de distancias de todas las celdas entre los procesos disponibles.
- Sin embargo, cuando aumenta el número de individuos, se reducen las columnas y esta idea ya no resulta viable. Cada nueva distancia requiere de un cómputo significativo

realizando los cálculos. Por eso, es mejor juntar a todos los procesos para calcular las distancias de forma escalonada, dividiendo los individuos del *cluster* para encontrar la distancia mínima (simple) o máxima (completa).

La segunda estrategia se basa en esto, por lo cual el *worker* con la fila a actualizar tiene en cuenta cuantos *clusters* tiene el algoritmo en el momento de la actualización. Si es mayor a la mitad, el cálculo de las nuevas distancias se reparte entre los procesos. En caso contrario, el *worker* envía a todos los *workers* disponibles la misma celda, para que se divida la búsqueda de la menor o mayor distancia.

La tercera estrategia no tiene en cuenta esto, siempre se dividen las celdas de las nuevas distancias, pero esta vez añadiendo procesos extra que se encargan exclusivamente de realizar estos cálculos. Además de los procesos *workers* que trabajan en el bucle principal del algoritmo.

3.2.2. K-Medias

De manera similar al algoritmo anterior, K-medias pertenece al aprendizaje no supervisado. Esta vez se aplica un valor K sujeto a una asignación flexible según nuestros criterios. Al contrario que el algoritmo jerárquico aglomerativo, no se utiliza una matriz, y solo se usa distancia por centroides. Las estrategias para este algoritmo son las siguientes:

1. Reparto estático. Dividir la población entre los *workers*, como muestra la Figura 3.6.
2. Reparto dinámico. En cada iteración del algoritmo, el *master* envía individuos a los *workers*. Cuando finalizan el procesado, envían de vuelta la agrupación y esperan a recibir otros individuos.

La primera idea es la más simple y la que -a priori- parece más prometedora en un primer momento. El *master* se encarga de generar los centroides de manera aleatoria, eligiendo K individuos al azar, sin repeticiones. Mediante *broadcast* los *workers* reciben estos centros, y

con conexiones punto-a-punto se recibe la población dividida sin intersecciones. Cada *worker* se encarga de una subpoblación.

El siguiente proceso se repite hasta que el *master* envíe un mensaje de finalización, es decir, no cambien los centros:

- Los *workers* calculan la asignación de sus individuos. Además, calculan la suma de distancias de los individuos a sus centros y el número de individuos asociado a cada *cluster*. Estos valores los envían al *master*.
- El *master* recibe estos valores y calcula los nuevos centroides. Manda un mensaje a todos los *workers*.
 - *CentroidesNuevos*, si los centros cambian. Se actualizan los centros.
 - Finalización, en caso contrario.

Después de implementar la primera opción, reparto estático, no es una buena idea implementar un reparto dinámico. El constante flujo de mensajes aumenta el tiempo de ejecución y la población a categorizar no cambia, por lo que distribuir la población una única vez en toda la ejecución es más eficiente. Lo único que cambia en cada iteración es la asignación de los individuos. Así, el algoritmo finaliza cuando la asignación no varía entre iteraciones (centros de los clusters). El algoritmo se repite *iter* veces, valor que depende de la asignación de los individuos y los centros de los *clusters*. Si además de parar la ejecución al recibir todos los individuos, se para de nuevo para reenviarlos al final de cada iteración, perdemos *iter* veces más tiempo al enviar los individuos a los procesos *workers*.

Como se comentó en el capítulo anterior, este algoritmo se tiene que repetir varias veces para encontrar la mejor asignación, el óptimo general. A su vez, también hay que variar el valor de K para buscar la mejor asignación para los individuos. Para lograr este objetivo se puede enfocar de dos diferentes formas:

1. Aplicar la implementación anterior con varios bucles. El externo para variar el valor de K y el interior para repetir el algoritmo.
2. Ejecutar en cada *worker* el algoritmo sin mejoras. Los *workers* ejecutan la búsqueda con valores distintos de K y el *master* se encarga de almacenar los mejores resultados.

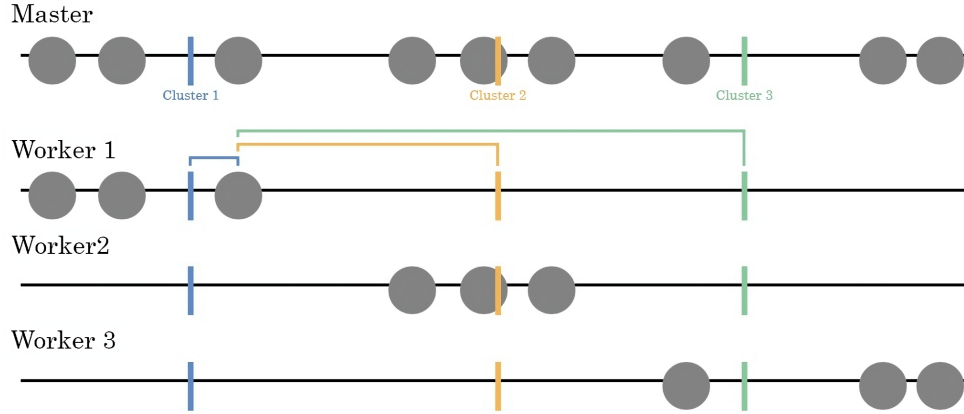


Figura 3.6: División de las poblaciones entre los procesos *workers* en la primera estrategia del algoritmo K-Medias

Como muestra la Figura 3.7, ambas ideas tienen el mismo coste temporal, sin contar el overhead (tiempo consumido para la comunicación de mensajes entre procesos). Pero, la segunda opción, al tener en cada proceso una copia de la población entera, tiene mayor complejidad espacial. Este aumento del uso de memoria para poblaciones elevadas provoca que no sea viable ejecutar esta estrategia en ejecuciones con un elevado número de procesos.

$$O\left(\left(T * \frac{N * K}{M}\right) * \text{Rep} * K_{\max}\right) \approx O\left(\frac{(T * (N * K)) * \text{Rep} * K_{\max}}{M}\right)$$

T = número de iteraciones en el algoritmo de K-Medias
 N = número de individuos en la población
 M = número de procesos *workers*
 K = número de clusters
 Rep = repeticiones para buscar el óptimo general
 $K_{\max}$ = valor máximo de K en la búsqueda.

Figura 3.7: Comparación temporal de las estrategias de búsqueda del óptimo general para el algoritmo de K-Medias

3.2.3. K-Vecinos más cercanos (KNN)

Al contrario de los algoritmos anteriores, KNN (K-Nearest Neighbors en inglés) pertenece al aprendizaje supervisado, por lo que necesita una población ya categorizada para poder agrupar los nuevos individuos. Al igual que el algoritmo K-Medias, y como menciona su nombre, esta técnica de clustering utiliza una variable K , que representa los vecinos que se van a utilizar para categorizar los individuos.

Aplicando una cola de prioridad de máximos para el cálculo de los K vecinos más cercanos, reducimos la complejidad del algoritmo. Al recorrer la población categorizada, se compara con la cima de la cola. Si la distancia a comparar es menor que la cima, se elimina la cima y se introduce la nueva distancia. Los valores de la cola se mueven con la restricción de prioridad, y al finalizar la búsqueda en la población se cuentan los elementos de la cola para saber qué *cluster* se repite más.

Es importante actualizar la población conforme se van prediciendo los valores, para tener más puntos de referencia. Si no actualizamos la población, la agrupación de los individuos puede variar de forma significativa. Aunque es menos precisa a la hora de predecir, el algoritmo es más rápido, ya que, al no aumentar la población categorizada, no tiene que recorrer un individuo más conforme se categoriza la población a predecir. Al contar con una población extra (la categorizada) se proponen dos estrategias posibles:

1. Dividir la población categorizada entre los *workers* (ver Figura 3.8a)
2. Dividir la población a predecir entre los *workers* (ver Figura 3.8b)

Si dividimos la población categorizada (primera estrategia), los *workers* cuentan con menos cómputo en cada individuo. Comparan el individuo a predecir con su subpoblación, y el *master* tiene una mayor carga de trabajo, al recibir los K vecinos de cada *worker* y tener que ver los K mejores de todos los individuos recibidos. Mientras que el *master* comprueba las distancias recibidas, los *workers* operan en la siguiente iteración, enfocándose en el próximo individuo a predecir. Si se actualizan los individuos, el *master* reparte de

forma equitativa los individuos que se categorizan. En cada iteración envía el individuo categorizado a un *worker* distinto, utilizando un contador con el *id* del *worker* respectivo y aplicando la operación módulo con el número de *workers* ejecutados.

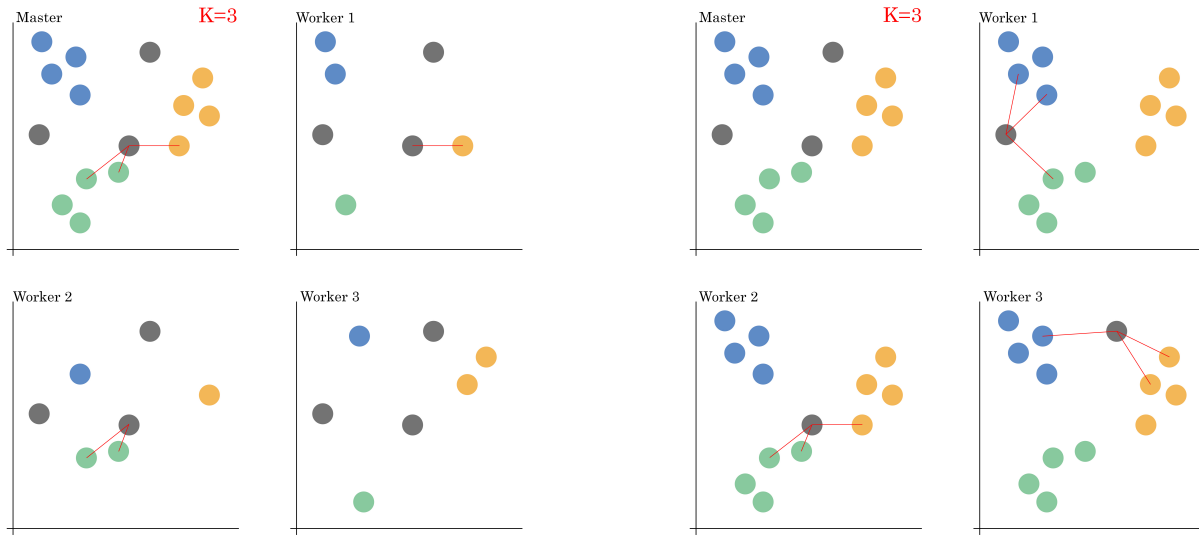
Con la división de la población a predecir (segunda estrategia), cada *worker* realiza un cómputo equitativo, pero predicen menos individuos. El *master* no realiza tantas tareas como en la estrategia anterior, solo recibe la categorización de los *workers*. El proceso de actualizar la población se puede realizar de varias formas, ya que todos los *workers* comparten la población categorizada. Una posible solución consiste en enviar, en cada iteración, el individuo categorizado a todos los *workers*. O en vez de actualizar en cada iteración, enviar cada X iteraciones los X nuevos individuos categorizados.

El coste espacial depende de los tamaños de las poblaciones.

- La primera estrategia, al dividir la población inicial, es más eficiente cuando esta población inicial es mayor que la población de predicción.
- En su contraparte, en la segunda estrategia, al dividir la población a predecir, tiene un mejor rendimiento con poblaciones de predicción mayores.

El proceso *master* (ver la Figura 3.8) tiene el *dataset* completo. En la primera estrategia (gráfico de la izquierda) divide la población categorizada (individuos representados con colores azul, verde y amarillo) entre los procesos y mantiene una copia en todos los procesos de la población a predecir (individuos representados en gris). La segunda estrategia (gráfico de la derecha) tiene una copia de la población categorizada en todos los procesos y una subpoblación de la población a predecir.

En este algoritmo también hay que realizar una búsqueda del mejor valor para K . Sin embargo, al contrario que en K-Medias, no hace falta repetir varias veces el mismo algoritmo, ya que el proceso es determinista. Es decir, con la misma población, siempre se obtiene la misma predicción. Aunque puede llegar a cambiar dependiendo del orden de categorización de la población a predecir.



(a) División de la población categorizada

(b) División de la población a predecir

Figura 3.8: Estrategias para paralelizar el algoritmo KNN

No hay que repetir el mismo algoritmo varias veces, pero puede llegar a ser útil variar el número de vecinos (valor de K). Las mejoras de esta búsqueda son las mismas que en el algoritmo anterior:

1. Aplicar alguna de las dos implementaciones comentadas anteriormente, con un bucle que varíe la variable K .
2. Ejecutar en cada proceso *worker* el algoritmo sin mejoras. El *master* se encarga de almacenar los mejores resultados.

3.3. Aprendizaje por refuerzo

Los algoritmos de este tipo de aprendizaje actualizan iterativamente las estimaciones de calidad de las acciones permitidas en el entorno de desarrollo, y pueden ser almacenados en una tabla o aplicar una red neuronal.

3.3.1. Q-Learning

El algoritmo Q-Learning es el más básico del aprendizaje por refuerzo. Las estimaciones de las mejores acciones para cada estado se almacenan en una Q-Table representada como una matriz en la que cada fila es un estado, y las columnas son las acciones disponibles.

Este algoritmo tiene numerosas aplicaciones. Nos centramos en la técnica de minimizar las acciones, para llegar desde una celda origen a un destino. El laberinto tiene un tamaño y semilla variable por parámetros de inicialización. Para lograr su objetivo dispone de acciones de movimiento en los dos ejes cardinales: norte, sur, este y oeste. No puede atravesar ni situarse en un muro del laberinto, y para que el agente aprenda a moverse por el laberinto y llegar a la meta hay que fijar unas recompensas:

- Si se choca con un muro castigamos al agente con valores altos para que no añada movimientos innecesarios para alcanzar su objetivo.
- Al moverse, el agente recibe un castigo pequeño para que aprenda a minimizar las operaciones.
- Al llegar a la meta le damos una recompensa alta, para que aprenda llegar a la celda destino.

Con estas recompensas, el agente aprende a llegar a la meta minimizando las acciones ejecutadas. El código que genera los laberintos ha sido implementado por @ChlouisPy en [github](#)²

Antes de enfocarnos en las implementaciones basadas en MPI, hay que comentar una mejora que se puede aplicar a él algoritmo básico de Q-Learning: realizar un preprocesado. Modificar la Q-Table, convirtiéndola en un array bidimensional, en el cual no se almacenen las acciones que no deseamos que realice el agente, como puede ser chocarse con un muro, o eliminar estados innecesarios como situarse en un muro.

Esta mejora puede reducir el tiempo de cómputo, al no perder tiempo realizando acciones innecesarias para alcanzar su objetivo. Además, se reduce la complejidad espacial al

reducir el número de estados. Una desventaja es que añade otra estructura adicional (array bidimensional) para almacenar las acciones para cada estado.

Este preprocesado tiene complejidad cuadrática $O(4 * N^2) \equiv O(N^2)$, siendo N el número de filas y columnas. Recorre toda la matriz, comprobando para cada celda si no es un muro, y, en caso afirmativo, itera en las cuatro direcciones permitidas para almacenar las acciones disponibles para la celda actual (estado). Con tamaños de laberintos pequeños no hace falta paralelizar el preprocesado, porque no se consigue reducir el tiempo significativamente. Al emplear laberintos con más de mil filas y columnas sí se consigue reducir el tiempo de cómputo.

En los algoritmos del bloque anterior se necesitaba realizar una búsqueda para encontrar el óptimo general. En este algoritmo conviene realizar otra búsqueda, pero esta vez para encontrar combinaciones de los hiper-parámetros (α , γ , ϵ), los cuales son muy importantes para el desarrollo del agente en el entorno. Una mala configuración de éstos hace que sobre aprenda -o no aprenda- correctamente, generando bucles infinitos. Por este motivo es importante comprobar tanto las diferentes combinaciones de hiper parámetros, como cuáles funcionan correctamente en el entorno. La búsqueda en laberintos grandes es muy lenta, ya que hay que comprobar muchas combinaciones entre los hiper-parámetros y los episodios del entrenamiento. Por eso es más útil desarrollar el algoritmo Deep Q-Learning, que no tiene problemas con los estados al usar una red neuronal. Pero si queremos usar el algoritmo básico de Q-Learning, hay que realizar una búsqueda exhaustiva en el entorno ejecutando una cantidad significativa de combinaciones de hiper parámetros.

Para paralelizar esta búsqueda se ejecuta el algoritmo básico con el preprocesado mencionado en cada proceso *worker*. Hay que desarrollar una estrategia para que cada *worker* reciba una combinación de parámetros distinta, y así no haya repeticiones, perdiendo tiempo de cómputo. El *master* se encarga de repartir combinaciones de hiper-parámetros. Con una precisión previamente inicializada, el *master* aumenta un hiper-parámetro hasta llegar al 100%. Cuando llega a dicho límite, se reinicia la variable y se aumenta la siguiente. Este

proceso continua hasta llegar al 100% de todos los hiper-parámetros. Cada *worker* se encarga de una combinación recibida. Cuando termina el algoritmo, ya sea por bucle infinito o finalización correcta, envía un mensaje al *master* con la información de finalización y los hiper-parámetros usados.

Si el mensaje de finalización indica “bucle”, el proceso termina para evitar que se convierta en un proceso inactivo, pues dicha combinación no converge hacia el objetivo. Estos bucles se detectan en el entrenamiento o la evaluación. Hay un bucle en el entrenamiento si un episodio tarda más de X segundos en finalizar. En la evaluación se comprueba teniendo en cuenta los últimos cuatro estados visitados, pues avanza y retrocede constantemente.

$$Estados[0] == Estados[2] \text{ and } Estados[1] == Estados[3] \text{ and } Estados[0] != Estados[1]$$

Una vez realizada una búsqueda de las combinaciones de hiper-parámetros eficaces, se pueden emplearse para las siguientes mejoras:

1. Dividir el entorno (laberinto) entre los procesos.
2. Ejecutar el algoritmo en los *workers* y juntar las experiencias.

Al dividir el laberinto entre los procesos (primera mejora), cada proceso controla una zona, y se genera un flujo constante de episodios (iteraciones del algoritmo). Cuando un agente sale del dominio de un proceso, éste le manda un mensaje al proceso que controla esa parte del laberinto con la posición en la que entra. La Figura 3.9 muestra cómo se divide un posible laberinto entre los procesos, además de mostrar tres episodios con sus respectivos agentes (circulo en rojo). El *master* se encarga de iniciar a los agentes en la celda verde, y cuando sale de su dominio, envía la posición en el laberinto al respectivo proceso y genera otro agente.

Para garantizar el correcto funcionamiento, el *master* no puede recibir agentes de los *workers*, en caso contrario no se podría garantizar el flujo de nuevos episodios. En la ejecución, solo pueden existir M agentes en todos los procesos (siendo M el número de procesos

ejecutados), debido a que un proceso solo puede gestionar a lo sumo un agente. Cada proceso tiene su propio dominio, lo que provoca que la Q-Table se divida entre éstos. Aplicando el preprocesado comentado anteriormente se dividen los dos arrays bidimensionales (acciones y Q-valores).

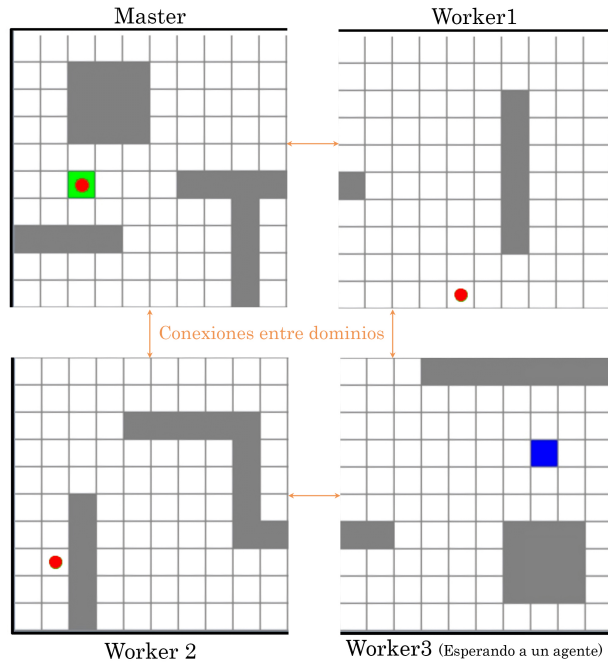


Figura 3.9: División del entorno de la primera estrategia del algoritmo de Aprendizaje por Refuerzo

Aplicando la estrategia realizada en la búsqueda de hiper-parámetros, consistente en ejecutar el algoritmo en varios procesos, se puede obtener la segunda estrategia. El *master* recolecta las experiencias de los *workers*, haciendo la media de los Q-valor obtenidos de los procesos, calculando así las mejores acciones para cada estado. Hay que tener en cuenta la posición de inicialización de los agentes en los procesos ejecutados. Si todos los procesos comparten el mismo punto de salida, los resultados serán parecidos a ejecutar el algoritmo en un solo proceso. Sin embargo, al cambiar el punto de origen, los Q-valores obtenidos al realizar las medias de las experiencias varían y se recorre más espacio en menos tiempo. Para lograr buenos resultados se asegura que al menos un proceso empieza desde el punto

origen, de otra forma no se podría garantizar que el agente haya aprendido a alcanzar el destino desde la celda origen.

3.3.2. Deep Q-Network

Este algoritmo utiliza redes neuronales para obtener la mejor acción para un determinado estado, eliminando así los problemas que tiene el algoritmo anterior. Con este método podemos abarcar entornos más complejos, y por eso se propone el juego de *Namco Pac-Man*, cuya implementación creamos desde cero para moldear a nuestro gusto la dificultad del entorno, así como facilitar el aprendizaje de la red neuronal. Nos centramos en obtener el mayor número de monedas antes de provocar una condición de finalización (ser comido por un fantasma o recoger todos los puntos). Para simplificar el entorno, no se desarrollan niveles en la ejecución, al igual de limitar la vida del agente a un único corazón, por lo que sí es comido una única vez se termina la ejecución. Antes de profundizar en el algoritmo de IA, explicamos cómo funciona y hemos realizado la implementación del entorno.

- Acciones disponibles. Como en el algoritmo anterior, son de movimiento. El agente y los fantasmas no pueden atravesar muros.
- Entorno. Laberinto con muros, del cual no se puede escapar.
- Objetos del juego.
 - Pac-Man: el agente que mueve el usuario. Su objetivo es comer todos los puntos.
 - Fantasmas: se mueven siguiendo unos objetivos en el laberinto.
 - Túneles: puntos que se conectan de manera toroidal para no salir del entorno.
 - Puntos (pellets en inglés): son las "monedas" que el agente tiene que recoger.
 - Puntos de energía (powers): si el agente consume uno, durante un periodo de tiempo es invencible y puede comer a los fantasmas.
 - Laberinto: entorno por el cual el agente y fantasmas se mueven.
- Condiciones de finalización. Ganar obteniendo todas las monedas del laberinto o perder si un fantasma come al agente.

Los fantasmas tienen una IA interesante, pues tienen sus propios estados y cada uno tiene unos puntos objetivos que siguen para intentar comer al agente. Cabe recalcar que estos puntos están estratégicamente colocados para que los fantasmas trabajen en conjunto para cerrar huecos y poder atrapar al agente. El movimiento para alcanzar los puntos objetivos es simple, cuando se encuentran en una intersección (punto en el mapa con un hueco a la izquierda o derecha con respecto a su dirección actual) eligen la celda que minimice la distancia con respecto al punto objetivo. Los estados de los fantasmas son los cuatro siguientes:

- Chase. Cada fantasma sigue unos puntos en movimiento.
- Scatter. Sigue un punto estático fuera del laberinto para dar vueltas en una determinada zona.
- Frightened. El agente puede comerlos, se mueve de manera aleatoria al llegar a una intersección.
- Eaten. Han sido comidos y se encuentran en su casa esperando a salir. (Implementado de forma que espera 3 movimientos del agente para salir)

Los estados iteran con una secuencia principal [*Scatter*, *Chase*]. La ejecución empieza con *Scatter* para que los fantasmas, al salir de su casa, se dirijan a sus zonas asignadas. Al ejecutar el agente treinta acciones, el estado cambia a *Chase* y se mantiene así sesenta acciones, volviendo a repetirse la secuencia. Si el agente come un punto de energía, los fantasmas interrumpen su estado actual para pasar al estado *Frightened* en el que están treinta acciones siendo vulnerables. Si el agente colisiona con un fantasma en este estado, es comido, pasando al estado *Eaten*. Al finalizar este estado vuelven al inicio de la secuencia principal.

En el estado *Chase* los fantasmas se mueven de la siguiente forma:

- Blinky (Rojo): Persigue directamente al agente.

- Pinky (Rosa): Persigue la celda cuatro posiciones adelantadas a donde apunta el agente. Si el agente mira hacia arriba, también añade cuatro celdas hacia la izquierda.
- Inky (Azul): Persigue una celda en concreto que se calcula de la siguiente forma. Primero se calcula una posición como lo hace el fantasma rosa, pero en vez de cuatro celdas, se hace con dos. El objetivo se calcula al añadir el vector de distancia de la posición del fantasma rojo a esta posición.
- Clyde (Naranja): Si está a ocho o más celdas de distancia del agente, lo persigue. En caso contrario sigue su objetivo del estado Scatter.

El laberinto (mapa del entorno) se almacena en un fichero de texto, para representar las celdas vacías, muros, puntos, o puntos de poder con números enteros (0, 1, 2 y 3 respectivamente).

Al igual que en el algoritmo *Q-Learning* la fase de entrenamiento es crucial, pues modifican los valores de la red neuronal para que el agente tome las mejores decisiones en cada estado, y terminar la ejecución sin perder. El entrenamiento se puede realizar de varias formas.

Si mantenemos el mismo estado inicial, el agente empieza siempre en el mismo punto, y depende mucho de los hiper-parámetros, además de la aleatoriedad. El agente empieza a investigar el entorno de manera aleatoria, y es muy probable que los fantasmas alcancen al agente bastante rápido sin explorar en profundidad el entorno. Por eso es mejor añadir varios estados iniciales para que pueda investigar el entorno de manera más eficiente. Hay que tener en cuenta que los estados iniciales tienen que ser puntos accesibles desde el estado inicial original. El agente no puede saltar a otras celdas sin coger los puntos del laberinto. La Figura 3.10 muestra un estado accesible y otro inaccesible con la misma posición del agente. El estado accesible ha recogido los puntos del laberinto, así como posicionado correctamente los fantasmas, en su contraparte el estado inaccesible no ha recogido los puntos simulando una acción de salto por parte del agente. Si entrenamos con puntos aleatorios sin cambiar

el estado del entorno, este entrenamiento no habrá surtido efecto, pues son estados que el agente no va a alcanzar nunca.

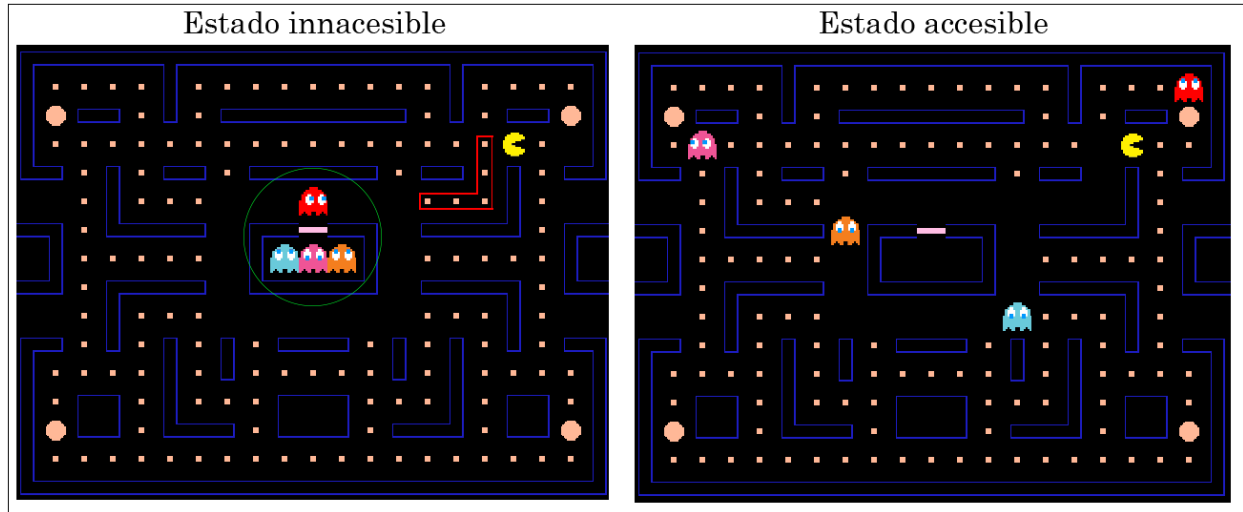


Figura 3.10: Tipos de estados del entorno en el algoritmo DQN

Como se comentó anteriormente, este algoritmo usa redes neuronales para aprender a ejecutar la mejor acción para un estado dado. Se van a aplicar las mejoras que se comentan en la Sección 3.5 de redes neuronales. En este caso, no se pueden aplicar las mejoras del algoritmo anterior, pues no es una matriz que se pueda dividir el trabajo, si no una red neuronal cuyos pesos varían al ejecutar acciones en estados.

3.4. Algoritmos Evolutivos

Los algoritmos evolutivos son sencillos de paralelizar. Trabajan con poblaciones de individuos que evolucionan a lo largo de las generaciones. Los individuos se someten a operaciones para producir nuevas generaciones. Estas operaciones de cada método son independientes, pues se puede dividir el cálculo entre varios procesos. Los métodos son las siguientes:

1. **Inicialización.** Dados los parámetros iniciales se crea la población con los individuos deseados. Hay diferentes tipos, con sus respectivas características.

- **Binarios.** Estos individuos son fáciles de inicializar, pero ralentizan la comunicación entre procesos, debido al gran elevado número de bits que es necesario enviar. Sin embargo, se puede enviar el número con su representación real en lugar de enviar todos los bits.
- **Reales.** Al igual que los binarios son fáciles de inicializar, pero esta vez son más portables, al usar la *base 10* como representación de los números, en vez del sistema binario (0's y 1's).
- **Árboles.** Más lentos para inicializar y difíciles de tratar. Se usan punteros y aumenta la complejidad al gestionarlos.

2. **Evaluación.** Este es la parte del algoritmo que más tiempo de ejecución puede llegar a consumir. Varía dependiendo del tipo de individuo. Como su nombre indica, evalúa a todos los individuos dependiendo de una función de *fitness*, que puede ser desde una fórmula matemática hasta una ejecución de un algoritmo en un entorno.
3. **Selección.** Se seleccionan a los individuos para una nueva generación. La aleatoriedad predomina en este método, y dependiendo de la estrategia escogida se puede dar más o menos probabilidad a los más aptos.
4. **Cruce.** Con una probabilidad dada, los individuos se cruzan para introducirlos a la nueva generación. Normalmente tendrán un mayor coste temporal que el método anterior, pues hay que realizar modificaciones en los individuos para realizar el cruce.
5. **Mutación.** Igual que el cruce, tiene una probabilidad para mutar. Normalmente es un poco más veloz que el cruce, debido a las estrategias implementadas y la probabilidad de mutación suele ser menor a la de cruce, provocando una menor tasa de ejecución en esta parte.

En este trabajo se desarrollan los siguientes problemas a optimizar para los tres tipos de individuos implementados:

1. Los individuos binarios, tienen un intervalo y una precisión como variable de inicialización. Queremos calcular el valor máximo o mínimo para ciertas funciones matemáticas. Los valores *fitness* se calculan con la representación real del cromosoma, que varía dependiendo de la precisión que se le asigna al ejecutar el algoritmo. Por ello hay que convertir la información de binario a real.

Para estos individuos, el algoritmo se ejecuta bastante rápido, además de alcanzar el máximo global con menos generaciones que los otros dos individuos. La función de evaluación es lineal $O(N)$, pues convierte número binario a real para ejecutar la función matemática. Reducir su tiempo de ejecución es desafiante, por el poco coste temporal y el tiempo necesario para enviar/recibir los individuos.

2. Los individuos reales, se enfrentan a un problema de mayor complejidad, como es la minimización del retraso total obtenible de un número de aviones *NumAv*, en un aeropuerto con *NumP* pistas. Este problema se puede resolver con vuelta atrás, pero tiene un coste exponencial $O(2^N)$ siendo inviable para la mayoría de tamaños.

Los aviones tienen unos tiempos de llegada asignados a cada pista, además de contar con un tiempo de separación mínimo entre tipos de aviones para garantizar la seguridad de los pasajeros. Este tiempo viene condicionado por el tipo de avión que llegó antes y el que llegará. La figura 3.11 muestra cómo se calcula el valor *fitness* de un individuo. Tiene un coste cuadrático $O(NumAv * NumP) \equiv O(N^2)$. No obstante, dado que el número de pistas es menor que el número de aviones, el coste amortizado será lineal

```
fitness=0
for avion in aviones:
    for pista in range(pistas): # Calculamos TLA para cada pista
        TLA = max(TLA(vuelo_anterior) + SEP[vuelo_anterior][vuelo_actual], TEL)
    # Se asigna el vuelo actual a la pista con minimo TLA calculado
    fitness+=(menor_TLA-menor_TEL)^2
    # menor_TEL: menor TEL de ese vuelo con todas las pistas
```

Figura 3.11: Función de evaluación para los individuos reales en el algoritmo evolutivo

3. Los individuos de tipo árbol, han de encontrar una sucesión de acciones en un entorno para maximizar las celdas visitadas en una matriz. Poniendo en contexto, el agente de este problema se encarga de podar un jardín de tamaño $N \times M$ (toroidal, es decir, los extremos de la matriz están conectados) con acciones de movimiento, giro y salto. El individuo es representado en forma de árbol con expresiones (*Exp*). Cada expresión representa un nodo que puede ser una función, si es un nodo intermedio del árbol, o terminal si es una hoja (nodo del árbol sin hijos). Las expresiones terminales son:

- Izquierda. El agente gira 90° grados a la izquierda. *Devuelve* $(0, 0)$.
- Avanza. El agente avanza una posición en con respecto a su dirección, y poda la casilla destino. *Devuelve* $(0, 0)$.
- Constante aleatoria (i, j) . El agente no se mueve. Este terminal devuelve la constante haciendo el módulo con respecto al número de filas y columnas.

Las expresiones funciones tienen, al menos, un hijo, y son las siguientes:

- Suma(*Exp*, *Exp*): Ejecuta las dos expresiones. Devuelve la suma vectorial de los resultados de las expresiones.
- Prong(*Exp*, *Exp*): Ejecuta las dos expresiones. Devuelve el resultado de la segunda expresión.
- Salto(*Exp*): Salta a una nueva posición determinada por el desplazamiento que devuelve la expresión. Por ejemplo, si el desplazamiento devuelve $(4, 2)$ el agente saltará 4 casillas en su dirección actual y 2 casillas a la izquierda.

El coste temporal de este algoritmo viene dado principalmente por la función de evaluación. Ejecuta la simulación en la matriz hasta cumplir un determinado número de *ticks* (acciones realizadas), que es proporcional al número de filas y columnas de la matriz. Aunque las funciones de cruce y mutación también tardan en ejecutarse, debido que hay que garantizar un buen uso de los punteros, pues en caso contrario no

funcionaría el algoritmo de forma correcta.

Cada una de las siguientes estrategias MPI se pueden configurar para cada tipo de individuo:

1. Dividir la población en subpoblaciones. Al ejecutar el bucle principal se divide la población entre los procesos para agilizar el trabajo.
2. Modelo de islas. La población se divide entre los procesos y, en paralelo, ejecutan el algoritmo de manera secuencial.
3. Pipeline. La población se divide en subpoblaciones que evolucionan de manera independiente. Cada proceso se encarga de ejecutar una parte del algoritmo. Estos reciben una subpoblación de un proceso con menor ID (*rank*), la procesan y envían al siguiente proceso.

La primera estrategia, representada en la Figura 3.12, consiste en dividir la población entre los procesos. En el comienzo de la estrategia, el *master* recibe la población inicializada y evaluada de los *workers*, comenzando el bucle principal del algoritmo, en el cual:

- El *master* se encarga de hacer la selección, y enviarla dividida a los *workers*. Mientras que los *workers* procesan los datos recibidos, el *master* almacena el progreso de los mejores individuos de cada generación.
- Los *workers* reciben la selección, para cruzar, mutar y evaluar su parte. Al finalizar estos procesos mandan sus subpoblaciones al *master* para empezar la siguiente iteración.

Para el modelo de islas (segunda estrategia), se aplica la estrategia que se ha desarrollado anteriormente para otros algoritmos. Cada proceso ejecuta el algoritmo básico, dividiendo la población general en los procesos ejecutados. Con este modelo se reduce la comunicación entre los procesos, pues al contrario que la estrategia anterior, no se reserva funcionalidades

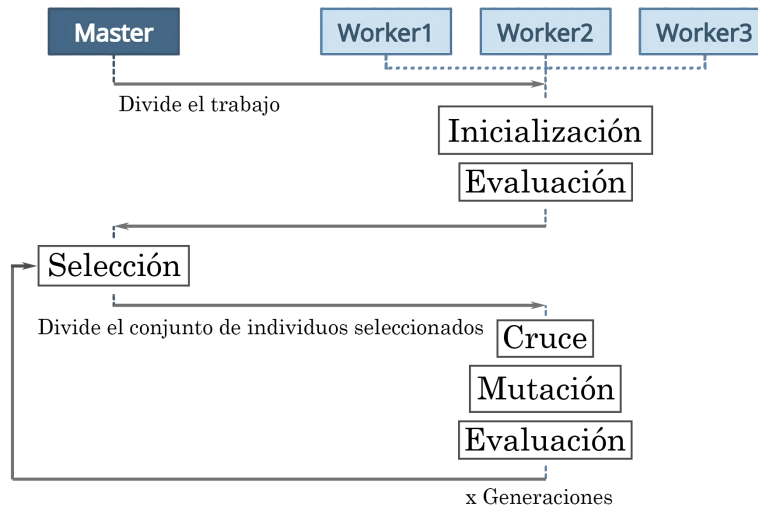


Figura 3.12: Primera estrategia (dividir la población) en el algoritmo evolutivo

para diferentes procesos, si no que cada proceso se encarga de ejecutar todas las funciones para que su subpoblación evolucione correctamente. Cada cierto tiempo los procesos se comunican, y reinician las poblaciones con los mejores individuos obtenidos en el proceso de comunicación. Los tipos de comunicación se representa en la Figura 3.13, y son los siguientes:

- Estrella. El *master* se posiciona en el centro y solo hay comunicaciones *Master-Worker* para almacenar los mejores individuos de cada subpoblación.
- En red. No hay proceso *master*, todos los procesos ejecutan el algoritmo. En cada comunicación, todos los procesos se comunican entre sí, mandando los mejores individuos para el reinicio de la población.
- En anillo. Igual que el anterior, pero esta vez no se comunican todos los procesos. La comunicación, como su nombre indica, se hace con topología circular, cada proceso se comunica con su predecesor y sucesor.

Segmentar el algoritmo entre los procesos (tercera estrategia, representada en la Figura 3.14), provoca un flujo constante de generaciones. Como hay cinco métodos principales se necesitan, al menos, cinco procesos, incluyendo al *master*, que se encarga de inicializar y

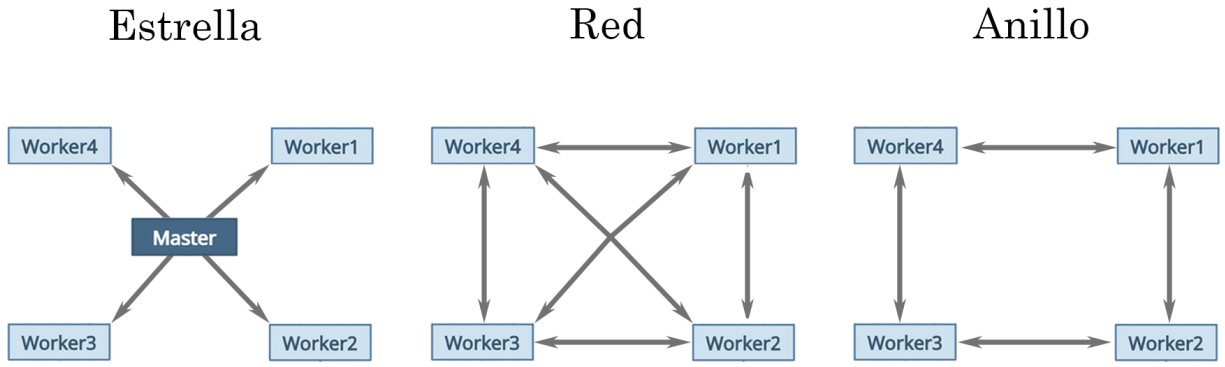


Figura 3.13: Segunda estrategia (modelo de islas) en el algoritmo evolutivo

evaluar las cuatro poblaciones distintas que se van a ejecutar al mismo tiempo. Al principio, todos los procesos *worker* están en espera de recibir una población para ejecutar sus operaciones. El primer *worker* se encarga de la selección y se despierta en la segunda iteración (*tick*) al recibir la primera población inicializada por parte del *master*. Una vez ha ejecutado las cuatro selecciones de las poblaciones que el *master* le envía, empieza a recibir las poblaciones del último *worker*. Los otros tres *workers* se encargan del cruce, mutación y el último de la evaluación, despertándose en la tercera, cuarta y quinta iteración (*tick*) respectivamente, por parte de su *worker* predecesor. El último *worker* finaliza una generación al enviar la evaluación de la población mutada al *worker 1*, encargado de realizar la selección. No hay conflicto con el *master*, ya que, como se mencionó anteriormente, éste crea cuatro poblaciones (terminando en la iteración cuatro, mientras que la primera evaluación del último *worker* se finaliza en la iteración cinco), y al finalizar su trabajo pasa a un estado de recepción de los mejores individuos. Si se ejecutan 100 generaciones, la población evolucionará en 104 *ticks* (iteraciones), donde los procesos trabajaran de manera conjunta en 96 *ticks*. Esto se debe a que el último *worker* se despierta al comienzo del quinto tick, y el primer *worker* termina de procesar datos al final del centésimo *tick*.

El tiempo de ejecución de esta estrategia es proporcional al número de generaciones ejecutadas multiplicado por el proceso que más tiempo tarda en ejecutarse. Por ello se puede reducir el tiempo de ejecución si comprobamos qué métodos tardan más, añadiendo

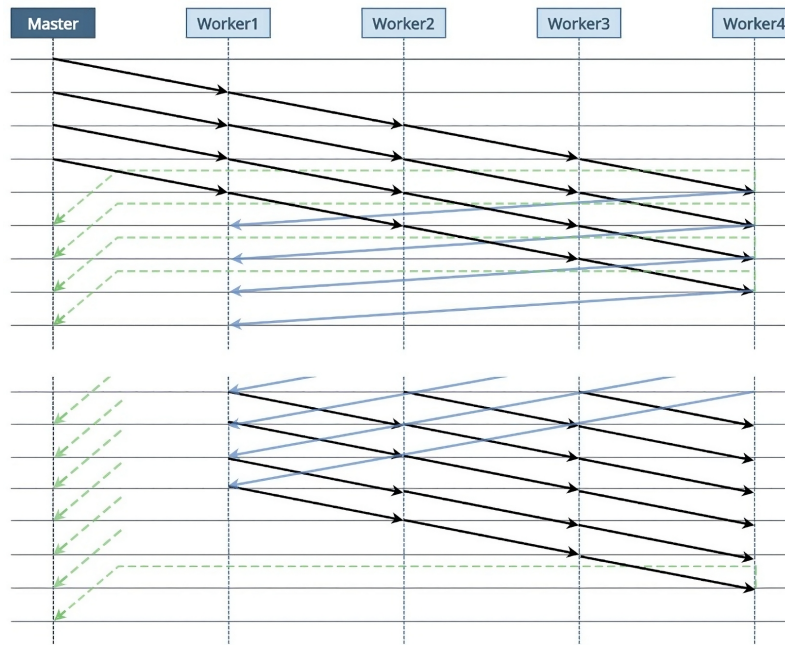


Figura 3.14: Tercera estrategia (pipeline) en el algoritmo evolutivo

más procesos para reducir la carga de trabajo. Por ejemplo, en los individuos reales y árboles, el método que más tiempo consume es la evaluación de individuos, por lo que conviene añadir más procesos para reducir así el tiempo de ejecución.

3.5. Redes Neuronales

Esta poderosa herramienta de aprendizaje supervisado está diseñada para reconocer patrones complejos y realizar diversas tareas. Aprende con un proceso iterativo de entrenamiento, ajustando las conexiones entre neuronas. Este proceso secuencial es complejo de paralelizar. Al finalizar una predicción, el modelo se tiene que actualizar propagando hacia atrás.

Nos centramos en la técnica de predicción. Para comenzar, vamos a crear una red neuronal que aprenda a predecir el Índice de Masa Corporal (IMC) de una persona. Cada individuo está formado por dos variables: la altura en centímetros y el peso en kilogramos. Para moldear la red neuronal a los individuos, la capa de entrada se estructura con dos

neuronas, una para cada variable, y la salida es el IMC, por lo que la capa de salida es una única neurona. La capa de entrada y salida no varían, pero la capa oculta se puede modificar libremente, aumentando el tiempo de la fase de entrenamiento.

Como en algunos de los algoritmos anteriores, necesitamos encontrar la mejor configuración de los hiper-parámetros. En las redes neuronales solo hay uno. La tasa de aprendizaje controla la magnitud de los ajustes a realizar en los pesos de las neuronas durante el proceso de entrenamiento. Específicamente, determina cuánto deben cambiar los pesos en respuesta al error cometido a predecir un individuo.

Por ello, diseñamos una estrategia para encontrar la mejor tasa de aprendizaje para una red neuronal en concreto. El proceso *master* envía intervalos de tasas de aprendizaje a todos los procesos *workers* ejecutados, para que éstos ejecuten el algoritmo y envíen el sumatorio de errores obtenidos en la predicción. La inicialización de los pesos normalmente es aleatoria, pero para hacer más igualitario el cálculo de los errores, todos los procesos inicializan la red neuronal con los mismos pesos.

Las estrategias MPI realizadas son las siguientes:

1. Pipeline. Como en el algoritmo anterior, pero esta vez con un flujo de mensajes bidireccional.
2. Dividir el trabajo entre los procesos.

Segmentar el proceso de entrenamiento puede llegar a ser beneficioso. Cada proceso se encarga de una capa de la red neuronal, siendo el *master* el encargado de enviar individuos de la población categorizada. El último *worker* controla la capa de salida, y con las etiquetas de los individuos, calcula el error y lo propaga hacia atrás. Para el correcto funcionamiento, hay que crear un buen diseño para tener un flujo constante de mensajes, los cuales pueden ser síncronos, es decir, que esperan a recibir los mensajes, o asíncronos, siendo estos últimos solamente admisibles en la etapa de recibir mensaje de una propagación hacia atrás anterior y enviar mensaje hacia adelante del individuo actual. La Figura 3.15 muestra dicho flujo y el trabajo de los procesos es el siguiente:

1. El *master* envía un número proporcional de individuos a los procesos en ejecución. Luego, antes de enviar otro individuo, entra en un bucle en el cual recibe el error de un individuo ya enviado, actualizando sus neuronas, y envía otro individuo. Para finalizar recibe el mismo número de errores (actualizando las neuronas) que individuos envió al principio.
2. El último *worker* solo recibe las predicciones y calcula el error.
3. Los *workers* de la capa oculta tienen un proceso más complejo. Primero, reciben un número de individuos proporcional a su *id*, los procesan y envían. Después entran en un bucle en el cual:
 - Reciben de la capa siguiente: los errores, actualizan sus pesos y lo propagan enviando lo a la capa anterior.
 - Reciben de la capa anterior: los nuevos individuos, procesan y propagan hacia adelante.

Al ser un proceso iterativo, en el cual el modelo va aprendiendo en la fase de entrenamiento, a primera vista, dividir la población entre procesos (segunda estrategia) no parece ser beneficioso para el correcto aprendizaje de la red. Sin embargo, en redes neuronales hay un proceso llamado *fine tuning*¹³ que consiste en entrenar una red neuronal, con unos pesos ya calculados. Basándonos ligeramente en esta técnica, podemos implementar una mejora en la cual dividamos la población inicial entre procesos y, en paralelo, ejecutamos la fase de entrenamiento. Una vez finalizadas, el *master* recibe los pesos de cada *worker* y hace la media. Cuanto más grande sea la red neuronal mayor será -a priori- el *speed-up*.

Las neuronas varían sus pesos para adaptarse a las variables recibidas. Para predecir un individuo, todas las neuronas trabajan en conjunto. Lo que supone que una neurona perteneciente a una red de menor tamaño tendrá más relevancia en comparación con una neurona en una red de mayor escala. Esto plantea un interrogante: ¿es posible paralelizar el trabajo de una red neuronal de gran escala y conservar -o reducir- el porcentaje de error al

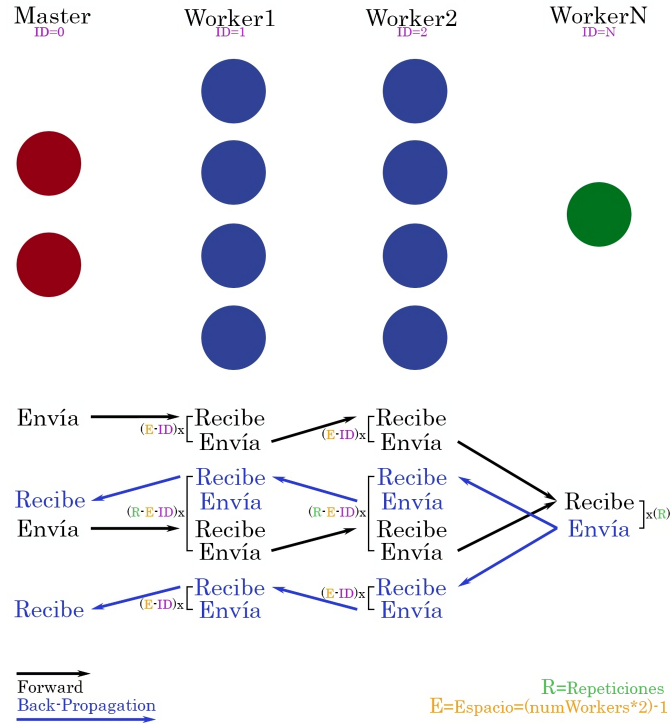


Figura 3.15: Primera estrategia en el algoritmo Red Neuronal

predecir individuos? Para ponerlo a prueba hay que probar varias estrategias de agrupaciones en la población que se va a utilizar para entrenar la red neuronal, además de tener en cuenta la inicialización de los pesos.

1. Misma población en todos los procesos.

- Si cada proceso tiene la misma población y los mismos pesos en la red, se reduce el tiempo de ejecución, pero no mejora la predicción. Sería como ejecutar el algoritmo sin paralelizar, pero con menos iteraciones, pues se ejecutan en los procesos la misma ejecución y la media no varía con respecto a los resultados obtenidos.
- Inicializando las distintas redes neuronales con pesos diferentes, puede predecir correctamente para este problema en particular, pero no tener unos buenos resultados de manera global o viceversa. Además, depende de la aleatoriedad, pues

el porcentaje de errores en una ejecución puede variar bastante con respecto a otra.

2. Diferentes poblaciones para cada proceso.

Esta estrategia suena mejor que la anterior. Al no haber intersección de poblaciones en los procesos ejecutados, los valores de los pesos se modificarán de diferente forma y puede que al hacer la media la red se estructure de forma que se obtenga un correcto funcionamiento. La inicialización de los pesos no provoca una gran diferencia, al contrario que mantener la misma población para todos los procesos. En cualquier caso, conviene probar ambas inicializaciones.

Capítulo 4

Estudio empírico

Después de diseñar e implementar las estrategias descritas en la Sección 3, llevamos a cabo un análisis exhaustivo para evaluar los tiempos de ejecución de cada una, así como contrastar resultados y extraer conclusiones

Primero, se ejecutan los experimentos en un ordenador de propósito general. Seguidamente se ejecutan las mejores implementaciones en un sistema distribuido con un número elevado de núcleos de CPU.

4.1. Entornos de ejecución

Para ejecutar los experimentos y comprobar el funcionamiento de las implementaciones, primero se ejecutan en un ordenador de propósito general. Este sistema computacional tiene las siguientes especificaciones:

- Procesador (CPU): *AMD Ryzen 7*, con 8 núcleos y 16 hilos, a 4.20 GHzs
- Memoria (RAM): 32 GB de *RAM DDR4*, permitiendo una amplia capacidad para manejar grandes volúmenes de datos en memoria. Característica fundamental para ejecutar algoritmos de IA que demandan una cantidad elevada de recursos.
- Tarjeta Gráfica (GPU): *NVIDIA GeForce RTX 3070* con arquitectura *Ampere*¹⁷, que cuenta con 5888 núcleos CUDA y 8 GB de memoria GDDR6. La arquitectura *Ampere*

es sucesora de la arquitectura *Turing* lanzada en 2020. Fue diseñada para brindar un mejor rendimiento, especialmente en aplicaciones de computación paralela.

- Placa Base: *X570 Gaming*. Soporta las tecnologías de conectividad de alta velocidad, garantizando el rendimiento y estabilidad del sistema en condiciones de carga elevada.

El sistema distribuido consta de tres ordenadores. Uno que funciona de Front-End y dos como nodos de cómputo, sumando entre estos últimos 128 núcleos de CPU y 256 GB de RAM. La figura 4.1 muestra la estructura del *cluster*.

El Front-End realiza la conexión remota con los otros dos ordenadores, situados en la Facultad de Informática de la Universidad Complutense de Madrid. Este ordenador no participa en el cómputo, solo mantiene los *scripts* (tipo de fichero de texto con el código escrito en un lenguaje de programación, en este caso Python) y lanza los experimentos sobre los nodos de cómputo. Durante las pruebas, cada proceso tiene un núcleo dedicado, por lo que el rendimiento de cada proceso no se ve afectado por otros procesos del sistema. Esto permite mayor precisión para evaluar el rendimiento del sistema, y las pruebas ejecutadas no compiten por los recursos de la CPU.

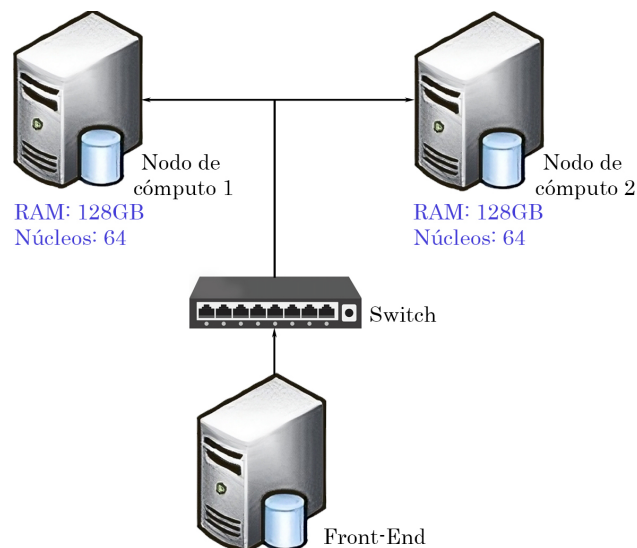


Figura 4.1: Estructura del sistema distribuido de la Facultad de Informática

Para realizar las pruebas se usan las funciones *open()* y *write()* de Python para almacenar los tiempos de ejecución en ficheros de texto. Los tiempos se miden con las funciones de tiempo de MPI, *MPI.Wtime()*.

4.2. Programas sencillos

Primero realizamos el estudio de los programas básicos descritos en la Sección 3.1, ordenación de arrays y multiplicación de matrices.

4.2.1. Ordenaciones

Las pruebas realizadas para estos algoritmos se realizan para el peor de los casos, es decir, un array de enteros sin repeticiones ordenado de forma decreciente. Cada algoritmo tiene que realizar el mayor número de comparaciones posible para ordenar el array de manera creciente. El resultado de cada experimento (tiempo de ejecución en segundos) es almacenado en un fichero de texto. Seguidamente se aumenta el tamaño del array para ejecutar el siguiente experimento, hasta llegar a *100.000* elementos.

4.2.1.1. Algoritmos de complejidad cuadrática

Debido al coste cuadrático de estos algoritmos, el incremento entre pruebas del tamaño de los arrays se obtiene de la siguiente forma:

- $[20 - 1,000) \rightarrow 20$ elementos.
- $[1,000 - 10,000) \rightarrow 250$ elementos.
- $[10,000 - 100,000) \rightarrow 1.000$ elementos.

SelectionSort es fácilmente paralelizable, pues para cada elemento se comprueba cuantos elementos en el array son mayores. Las estrategias implementadas utilizan el modelo *Master-Worker*. El *master* envía a cada proceso *worker* un elemento del array para que hagan las comparaciones y devuelvan el índice del elemento, junto con el número de elementos mayores que el recibido, y así el *master* se encarga de ordenar el array y enviar elementos sin procesar.

La Figura 4.2 muestra los tiempos de ejecución. En rojo el algoritmo sin mejora, y en verde y negro las dos estrategias ejecutadas con cinco procesos. Se puede apreciar una considerable reducción del tiempo de ejecución.

Al comparar las dos estrategias MPI, se obtiene que la primera estrategia es un 34 % más rápida que la segunda. Sin embargo, la segunda estrategia tiene una menor complejidad espacial, mostrando en la gráfica de la derecha, en negro, que la memoria no varía al aumentar los procesos ejecutados.

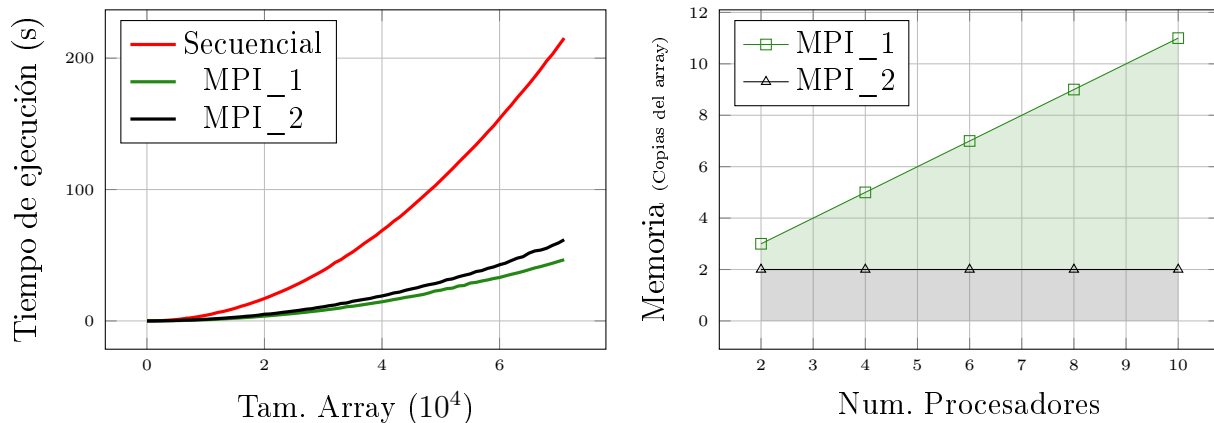


Figura 4.2: Tiempos de ejecución de las estrategias y su uso de Memoria para el algoritmo *SequentialSort* en ordenador de propósito general

Una vez comparadas las estrategias con el algoritmo secuencial, podemos comprobar el rendimiento frente a los algoritmos conocidos. La Figura 4.3 muestra que *SelectionSort* (la línea negra) es la ordenación que mejores resultados obtiene, y *BubbleSort* (línea roja) la que peores. *SelectionSort* es, aproximadamente, 3.5 veces más rápida al ordenar 70.000 elementos. La ordenación *SequentialSort* sin paralelizar, es incluso más rápida que dos de las más conocidas. Esto es debido a la simpleza de las operaciones aplicadas en la ordenación, pues solo hace N^2 comparaciones. En *BubbleSort* e *InsertionSort*, además de realizar comparaciones, modifican las posiciones de los elementos en el array, aumentando el tiempo de ejecución.

La estrategia MPI de *SequentialSort* que menos tiempo requiere (la primera) no obtiene

mejores resultados que la mejor ordenación sin mejoras (*SelectionSort*) hasta llegar a los cuatro procesadores, siendo un *20 %* más veloz. Para mostrar de forma más clara la diferencia de tiempos entre estas dos ordenaciones, se muestra la ejecución de la primera estrategia con cinco procesadores (*Sequential_MPI(5)*), obteniendo un *50 %* de mejora.

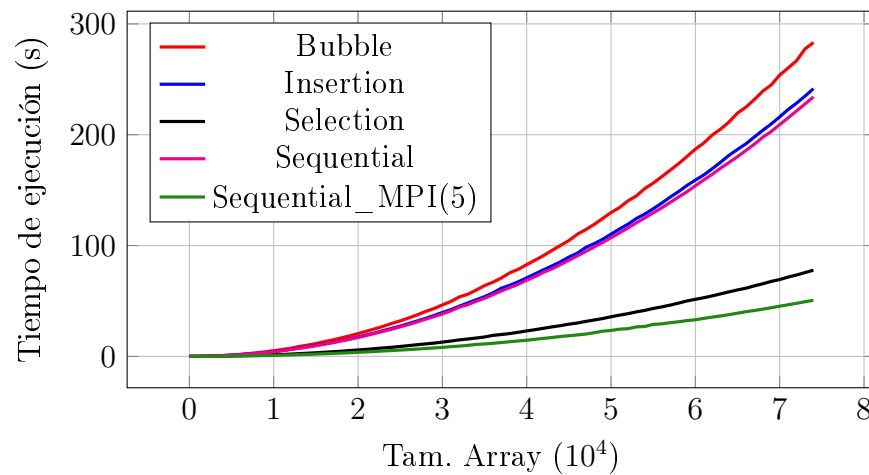


Figura 4.3: Tiempo de ejecución de los algoritmos de ordenación cuadráticos en ordenador de propósito general

4.2.1.2. Algoritmo *MergeSort*

Este algoritmo no tiene un coste tan elevado como los anteriores. La complejidad es logarítmica $O(N \log N)$ lo que provoca que se pueda aumentar el tamaño del array a ordenar. Para la estrategia implementada, no se aplica el modelo *Master-Worker*, sino que todos los procesos creados trabajan de manera equitativa. Como se dijo en la Sección 3.1, esta estrategia usa potencias de dos procesos para ordenar el array. En cada iteración los procesos se comunican con el más cercano, uno le envía su subarray ordenado y termina su ejecución (el de mayor *id* de cada pareja), mientras que el otro reordena los dos subarrays y continúa a la siguiente iteración.

En esta ocasión, la prueba realizada consiste en ordenar de manera creciente cuatro arrays de enteros inicializados de manera decreciente (peor de los casos), empezando con *25.000* elementos e incrementando esa misma cantidad entre los experimentos. Pese a tener

solo ocho núcleos en el ordenador de propósito general, se comprueba el rendimiento de la estrategia con 4, 8, 16 y 32 procesos. La Figura 4.4 muestra los resultados obtenidos en forma de histograma. Como la estrategia aplica ordenaciones cuadráticas en los subarrays al comienzo del algoritmo, no se obtienen buenos resultados con pocos procesos, debido al elevado tamaño del array a ordenar. Con dos procesos no reduce el tiempo de ejecución, lo duplica. El cómputo es equivalente a aplicar una ordenación cuadrática con la mitad del array a ordenar. No obstante, se puede apreciar una notoria reducción del tiempo de ejecución a partir de 16 procesos, llegando a tener un *speed-up* aproximado de 15.5. Es cierto que se podrían aplicar otras ordenaciones con menor complejidad para reducir más el tiempo, pero así se demuestra que en la computación de alto rendimiento se pueden obtener buenos resultados con estrategias no tan efectivas, pero bien paralelizadas.

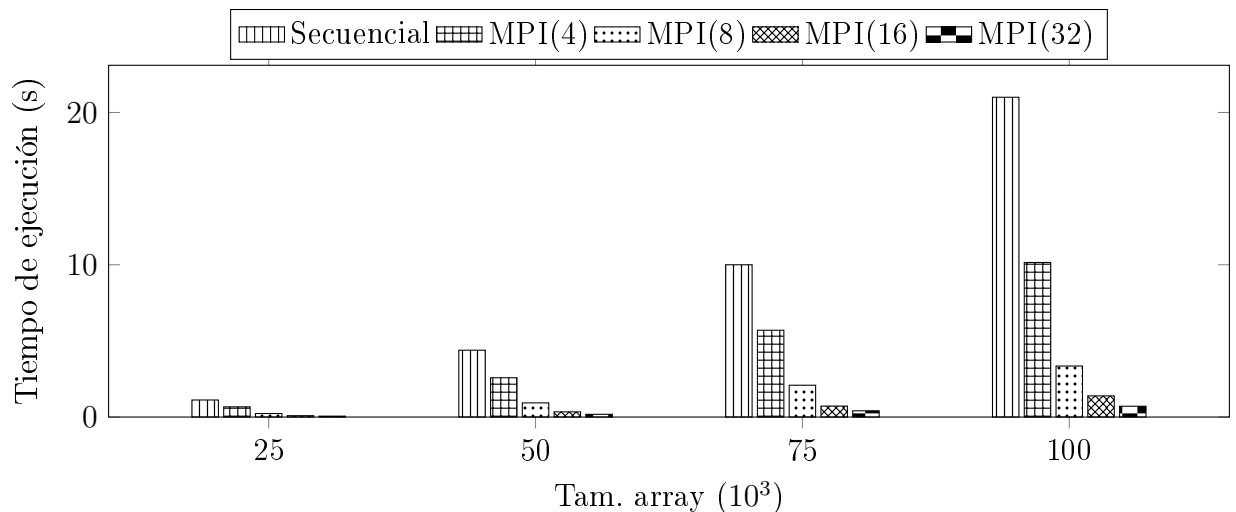


Figura 4.4: Tiempo de ejecución del algoritmo *MergeSort* en ordenador de propósito general

La memoria está optimizada, puesto que el array está dividido entre los procesos. Al terminar un proceso con la sincronización en mariposa comentada en la Sección 3.1, se termina la ejecución del proceso liberando memoria una vez ha enviado al proceso correspondiente su subarray ordenado.

Seguidamente, pasamos a comentar las pruebas realizadas en el sistema distribuido.

El algoritmo secuencial de *MergeSort* tarda unos 20.16 segundos en ordenar, de manera

creciente, un array de 100.000 elementos ordenados de manera decreciente (el peor de los casos). Por ello, realizamos un experimento para saber cuánto tiempo requiere la estrategia en ordenar un array con un millón de elementos. Las pruebas comienzan con 100.000 elementos, incrementando nueve veces su tamaño hasta llegar al millón de elementos. Estas pruebas se realizan con 16 , 32 , 64 y 128 procesos. La Figura 4.5 muestra los resultados obtenidos.

De manera secuencial, sin mejoras, el algoritmo requiere 20.16 segundos en ordenar 100.000 elementos, mientras que con 128 procesos requiere 0.16 segundos, obteniendo un *speed-up* de 125 .

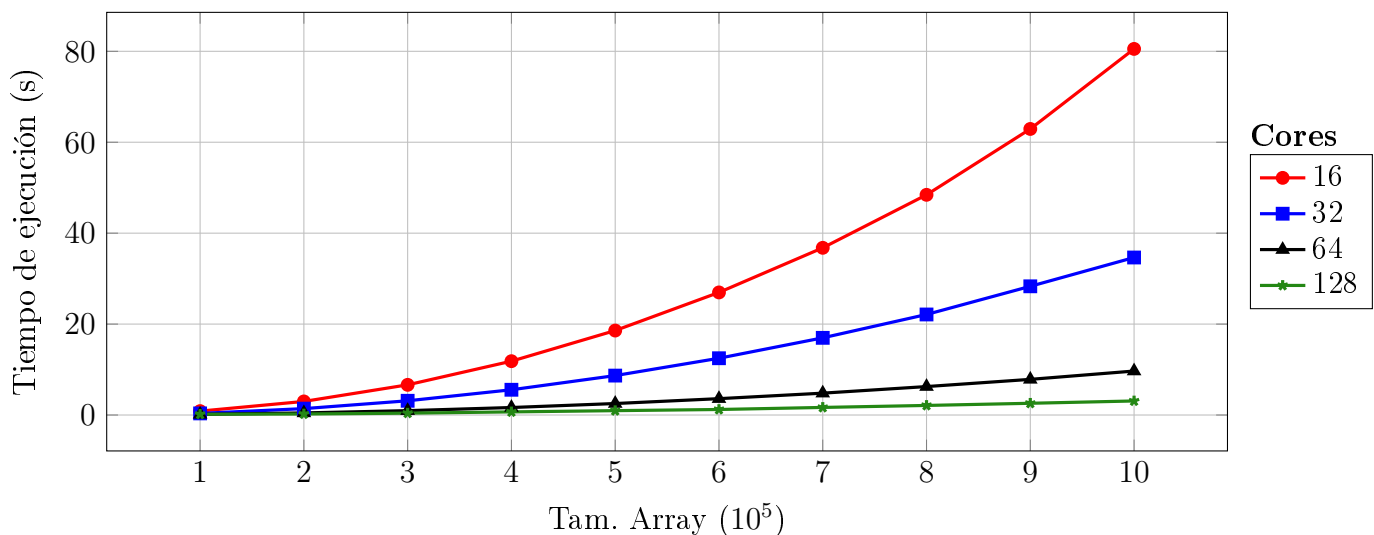


Figura 4.5: Tiempo de ejecución del algoritmo *MergeSort* en el Cluster

4.2.2. Multiplicación de matrices

Para este algoritmo, al contrario que los anteriores, no hay caso peor, pues siempre se ejecutan el mismo número de multiplicaciones para cualquier combinación de una matriz.

Las pruebas se realizan con una única matriz cuadrada de tamaño 1750 . Se genera de manera aleatoria con valores enteros que oscilan entre $[1-9]$, y es almacenada para usar en cada prueba. Inicialmente, la matriz empieza con diez filas y columnas, al finalizar una

prueba, se almacena el tiempo de ejecución y se incrementa el tamaño en diez, así hasta llegar a 1750 filas y columnas.

La distribución de tareas de los procesos se realiza mediante el modelo *Master-Worker*. El proceso *master* se encarga de enviar una matriz completa (B) y filas de la matriz (A) a los *workers* para que realicen el cálculo. Al finalizar el procesamiento envían de vuelta el resultado al *master* y esperan otras filas sin procesar, para poder, al final de la ejecución, formar entre todos la matriz final (C). ($A*B=C$)

Cada proceso necesita, al menos, una copia de una matriz completa. El uso de memoria es proporcional al número de procesos ejecutados. No hace falta tener las dos matrices porque el *master* se encarga de repartir filas para que vayan realizando el cálculo.

Seguidamente, se ejecutan los programas de multiplicación de matrices en el ordenador de propósito general. La Figura 4.6 muestra la ejecución del algoritmo secuencial, además de la estrategia implementada en la Sección 3.1, con 2, 4 y 6 procesos *workers*. Se puede apreciar la reducción de tiempo, aplicando la estrategia con los diferentes números de procesos, llegando a obtener un *speed-up* de 8.4 al ejecutar el algoritmo con seis *workers* en una multiplicación de 1750×1750 .

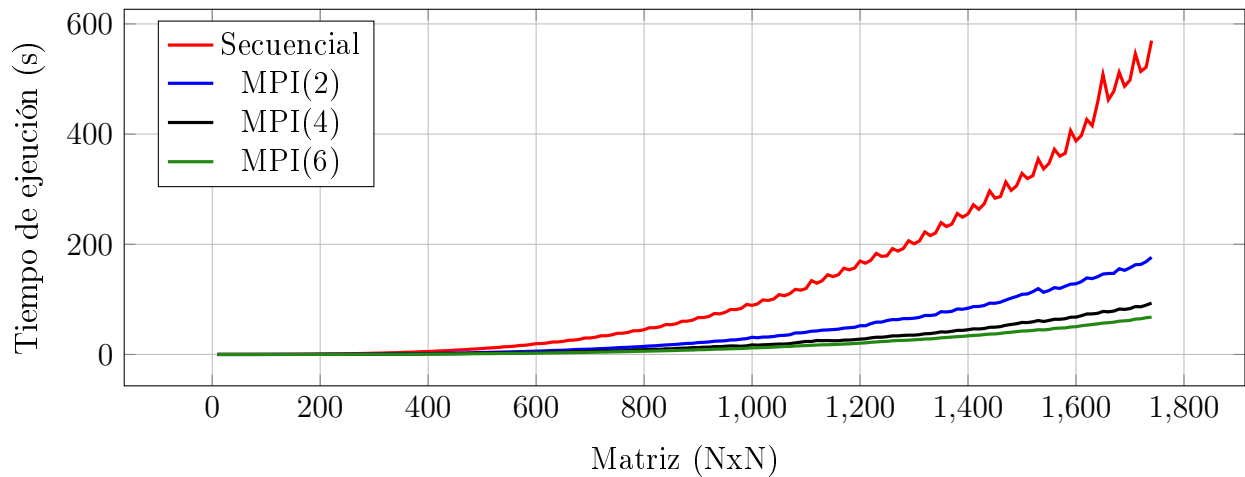


Figura 4.6: Tiempo de ejecución de multiplicación de matrices en ordenador de propósito general

Las oscilaciones en la gráfica se deben al incremento de diez elementos entre pruebas en un algoritmo con complejidad cúbica $O(N^3)$. Estas oscilaciones son más pronunciadas en la multiplicación sin paralelizar, pues el tiempo de ejecución es mayor.

En el *cluster*, al poder ejecutar un número elevado de procesos, se puede aumentar el tamaño de las matrices de las pruebas a ejecutar. Utilizamos *16*, *32*, *64* y *128* procesos para medir el tiempo que tarda el algoritmo con la misma estrategia que la prueba anterior. En este caso, comenzando con *500* elementos por fila, e incrementando ese mismo tamaño hasta llegar a una matriz de *5000* filas y columnas.

La Figura 4.7 muestra los tiempos de ejecución con los procesos y tamaños comentados en el párrafo anterior. No se puede apreciar, pero con una matriz de *1000* filas, ejecutar *128* procesos reduce el tiempo de ejecución hasta unos *1.06* segundos, logrando un *speed-up* de *84* con respecto a los *89.1* segundos del cálculo sin paralelizar. La comunicación entre procesos no es tan optimizada como en la estrategia de *MergeSort*, debido a que en esta implementación aplicamos el modelo *Master-Worker* y es posible que un proceso *worker*, al finalizar de procesar una fila, tenga que esperar a que el *master* esté libre (puede estar recibiendo y colocando otros datos recibidos de otro proceso) para recibir nuevos datos que procesar. En cada experimento, se pierden $(N/M)*T$ segundos en la comunicación entre procesos. Siendo N el número de filas de la matriz, M el número de *workers* y T el tiempo de comunicación.

4.3. Algoritmos de Agrupación

Las poblaciones que se utilizan en las pruebas de esta sección se han almacenado en un fichero para usar la misma población generada de manera aleatoria, con dos variables de entrada, es decir, individuos es un plano bidimensional. Los valores son delimitados en el siguiente intervalo $[-10, 10]$, ya que los valores no influyen en el cálculo de la distancia. Los tamaños de las poblaciones se incrementan dependiendo de la complejidad temporal de cada algoritmo. En sus secciones se especifica en profundidad.

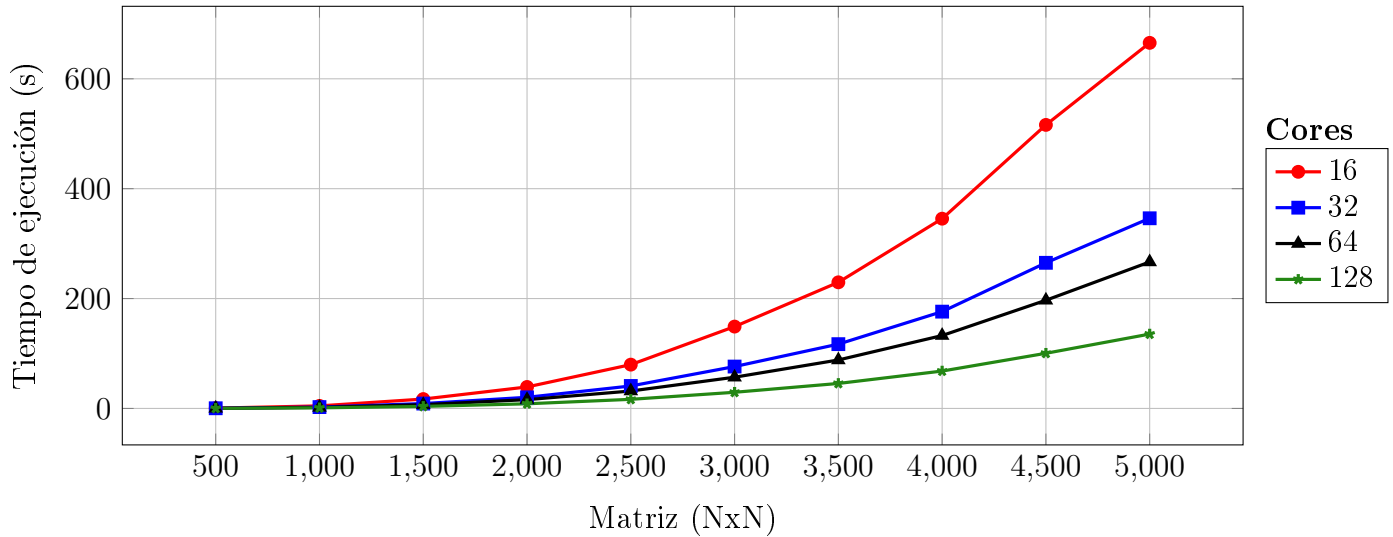


Figura 4.7: Tiempo de ejecución de multiplicación de matrices en el Cluster

Los tres algoritmos de esta sección se basan en el modelo *Master-Worker*. El *master* divide los datos de entrada para que los *workers* hagan el procesamiento. El *master* en cada algoritmo tiene las siguientes funciones:

- Jerárquico Aglomerativo. En cada iteración se encarga de gestionar qué procesos tienen que eliminar o actualizar las filas y columnas de la matriz. El *master* no tiene una copia de la matriz, mejorando así el uso de memoria al estar dividida entre los procesos *workers*.
- KMedias. Su función principal es comprobar la condición de finalización. En cada iteración recibe las asignaciones de los datos procesados de los *workers*, y si esta asignación no varía, se finaliza la ejecución.
- K-Vecinos más Cercanos (KNN). En las dos estrategias se encarga, de diferente forma, de actualizar las poblaciones de los *workers* para que haya más precisión a la hora de categorizar nuevos individuos.

4.3.1. Jerárquico Aglomerativo

De los tres algoritmos de agrupación, este es el más lento. Su bucle principal itera $N \cdot C$ veces, siendo N el número de individuos de la población y C el número de *clusters* deseados. En cada iteración, recorre una matriz entera para juntar dos *clusters*, los que se encuentren a menor distancia.

En este algoritmo, el cálculo de las distancias entre *clusters* es muy importante. Cada tipo genera diferentes agrupaciones, además de tener diferentes complejidades temporales. La distancia por *centroides* es la que menos tiempo consume, siendo constante, al solo importar los centros de los clusters. Mientras que *enlace simple* y *completo* tienen una complejidad cuadrática $O(N^2)$, recorriendo todos los individuos de ambos *clusters* para calcular la distancia. Además, hay que añadir el cálculo de la distancia entre individuos, que puede ser *Manhattan* o *Euclídea*, siendo esta última un poco más costosa computacionalmente que la primera.

Para mostrar la importancia de las distancias entre *clusters* y entre individuos, se realiza un estudio de los algoritmos sin aplicar ninguna estrategia de alto rendimiento (HPC). Con la población almacenada, se ejecutan las diferentes combinaciones de distancias (entre *cluster* e individuo), generando 4 tipos, pues entre *enlace simple* y *completo* solo varía almacenar la menor o mayor distancia entre clusters. Empezando con veinte individuos, y aumentando ese mismo tamaño hasta llegar a mil. A partir de este punto, es mejor incrementar en 250 individuos. La Figura 4.8 muestra los resultados de este experimento

Al principio no hay tanta diferencia, pero conforme aumenta el tamaño de la población, los tiempos de ejecución empiezan a distinguirse. Como muestra el círculo rojo, la distancia entre *clusters* por *centroide* no varía mucho usar una distancia *Euclídea* o *Manhattan* entre individuos. No obstante, aplicando *enlace simple* (o *completo*) es mejor usar distancia *Manhattan*. El cálculo de distancias entre individuos no usa potencias o raíces cuadradas, operaciones con un mayor coste que restas en valor absoluto. Cabe recalcar la diferencia de las distancias entre *clusters* por *centroide* y por *enlace simple* y *completo*. Al aumentar la

población a categorizar, aumentan los tamaños de los *clusters*, sobrecargando el cálculo de nuevas distancias.

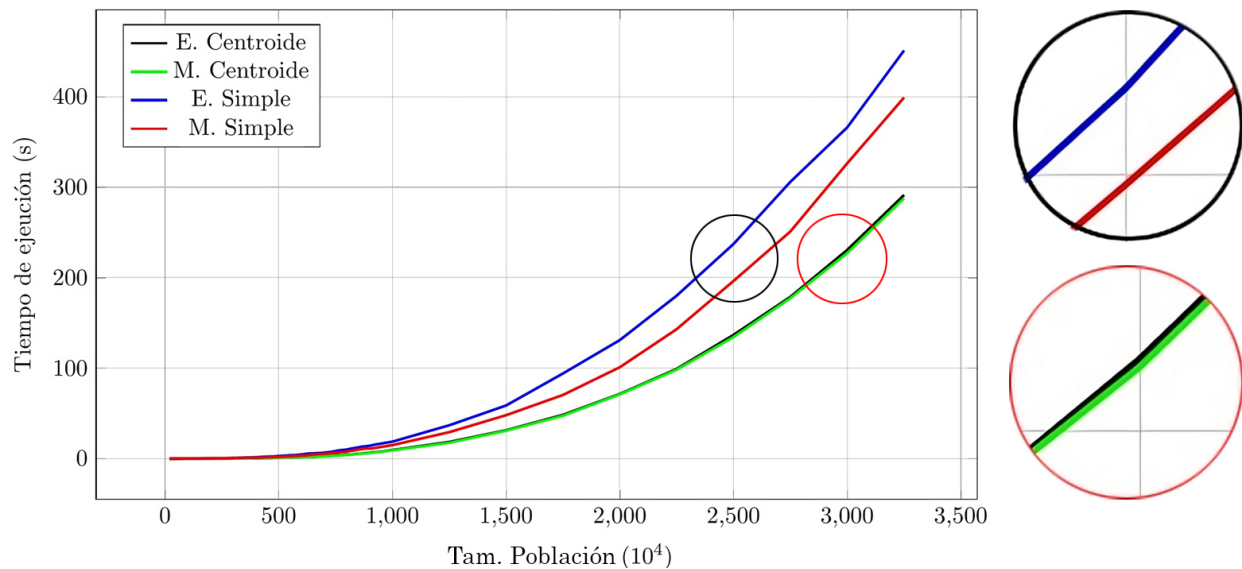


Figura 4.8: Tiempo de ejecución de las combinaciones de distancias en el algoritmo secuencial Jerárquico Aglomerativo

Una vez comprobado los tiempos de ejecución del algoritmo sin paralelizar, podemos pasar a las pruebas de las estrategias comentadas en la Sección 3.2.1. Primero, estudiamos la distancia entre *clusters* con menor tiempo de ejecución, por *centroide*. Ejecutamos, con 2, 4, 6 y 8 procesos la primera estrategia en tres diferentes poblaciones. No aplicamos la segunda y tercera estrategia, pues estas están diseñadas para las distancias por enlace *simple* y *completo*.

La Figura 4.9 muestra un buen rendimiento, reduciendo los tiempos de ejecución hasta con tamaños de poblaciones elevados. Para una población de 5000 individuos se consiguen los siguientes *speed-ups* [1.78, 2.89, 3.61, 4.21] utilizando 2, 4, 6 y 8 procesos, respectivamente. Los *speed-ups* no crecen en proporción a los procesos ejecutados. La estrategia implementada, para tamaños de poblaciones elevados, no es óptima. Si un proceso no elimina filas en muchas iteraciones, acumula muchos más datos que procesar que los demás procesos, provocando que los procesos con menos datos esperen para seguir con la ejecución.

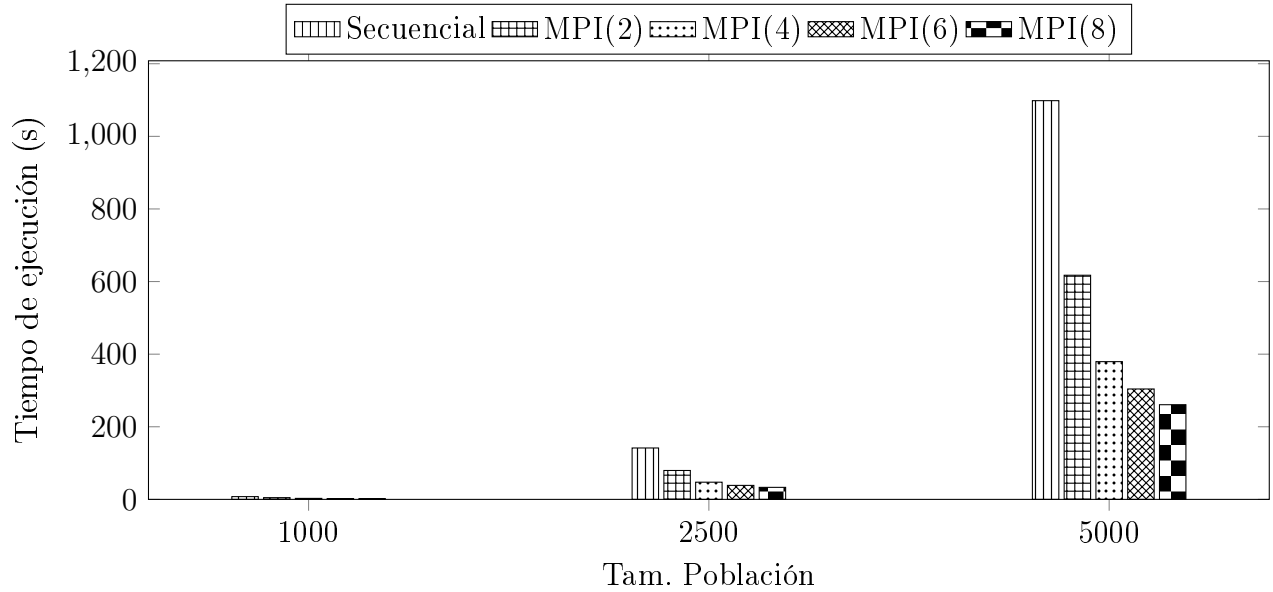


Figura 4.9: Tiempo de ejecución de la distancia entre *clusters* por *centroide* del algoritmo Jerárquico Aglomerativo en ordenador de propósito general

Ahora veamos el comportamiento de las estrategias para la distancia entre *clusters* con mayor complejidad, enlace *simple* o *completo*. Las pruebas se realizan con tamaños de poblaciones inferiores a las pruebas anteriores. Estos son los siguientes $[100, 200, 500, 1000, 1500, 2000]$, y se ejecutan las estrategias con cuatro procesos para comprobar el rendimiento. La tercera estrategia tiene el mismo rendimiento que la segunda, pero con más procesos. Reservar procesos únicamente para el cálculo de nuevas distancias no es eficaz, es mejor dividir entre los procesos activos (segunda estrategia). La Figura 4.10 muestra los resultados obtenidos del estudio. Aunque sí reduce el tiempo de ejecución, no se obtienen buenos resultados, pues el *speed-up* con 2000 individuos de población para la estrategia con mejores resultados es de 1.88. Al usar cuatro procesos, podemos concluir que los tres *workers* pierden mucho tiempo calculando las distancias en cada iteración. Es posible que, mediante la refinación progresiva de la segunda estrategia a través de un proceso iterativo de prueba y error, se logre reducir el tiempo de ejecución. No obstante, hasta el momento, no hemos logrado reducirlo más allá del tiempo actual.

Los resultados de la prueba anterior, con la complejidad del algoritmo indican que no

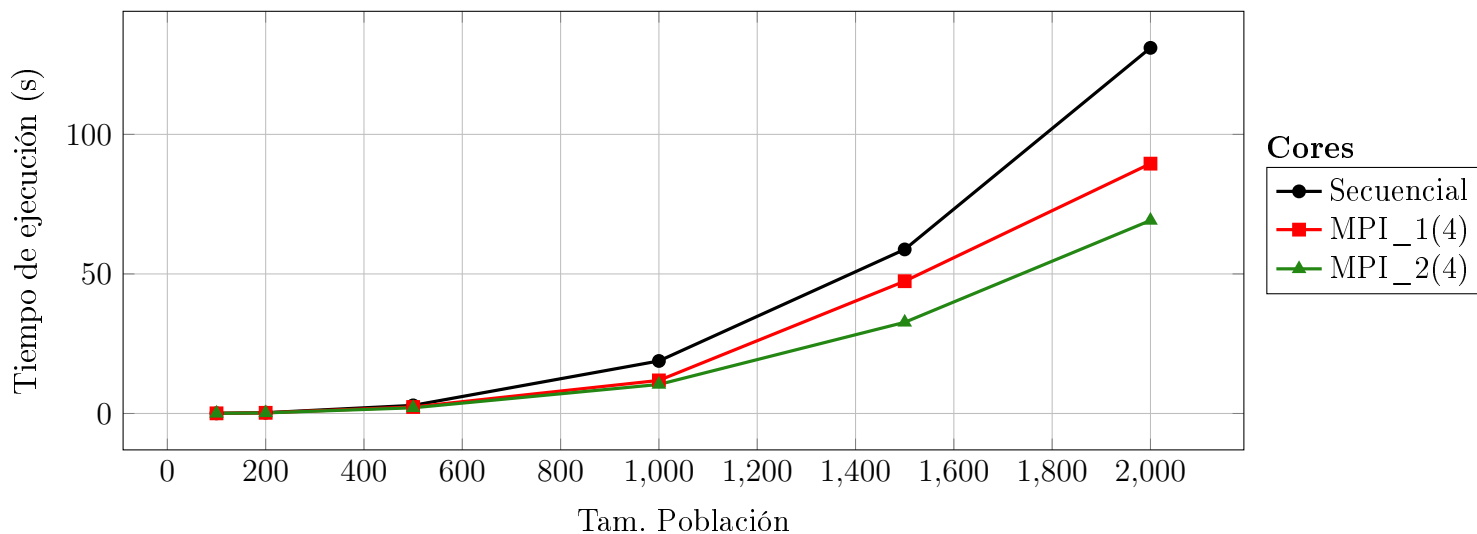


Figura 4.10: Tiempo de ejecución de la distancia entre *clusters* por enlace *simple* del algoritmo Jerárquico Aglomerativo en ordenador de propósito general

es viable probar las estrategias implementadas sobre estas distancias entre *cluster* en el sistema distribuido. Por este motivo, solo se prueba la distancia por *centroides* con tres grandes poblaciones. Los tamaños son los siguientes $[5000, 7500, 10000]$, y se prueban con 20, 50, 75, 100 y 128 procesos. La Figura 4.11 muestra los resultados, y concluimos que para agrupar tamaños de poblaciones elevados no conviene aplicar este algoritmo.

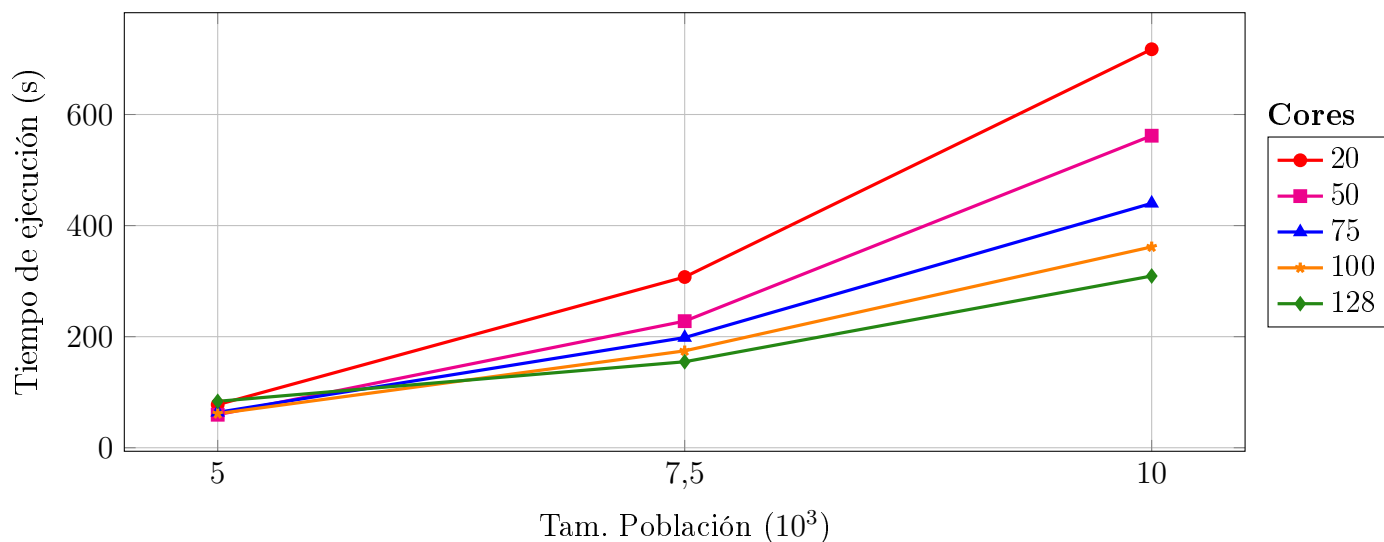


Figura 4.11: Tiempo de ejecución de la distancia entre *clusters* por *centroide* del algoritmo Jerárquico Aglomerativo en Cluster

O por lo menos las estrategias implementadas no dan resultados notorios, pues el *speed-up* entre usar 20 o 128 procesos en una población de 10000 individuos es de 2.32.

4.3.2. K-Medias

El algoritmo anterior no tiene ninguna variable que modifique el tiempo de ejecución (sin contar la distancia entre clusters). Esta técnica de agrupación tiene un coste temporal mucho menor que el aglomerativo, $O(N*K*iter)$ siendo N el tamaño de la población, $iter$ las iteraciones hasta que no cambien los centros y K el número de centros. ($N \gg K, iter$) K e $iter$ no son valores muy altos, por lo que la complejidad no llega a ser cuadrática. Cuanto mayor sea el valor de K , más tiempo va a consumir para realizar la asignación, pues cada individuo de la población es comparado con más centros. No obstante, dependiendo de la asignación de los individuos, una ejecución con más centros puede ser más rápida que otra con menos centros. Todo depende de la variable $iter$, es decir, si consigue llegar antes a la condición de finalización (que los centros no cambien entre dos iteraciones). La Figura 4.12 muestra precisamente este punto. Para dos poblaciones distintas, de 75000 y 100000 individuos aplicando $K=25$ centros (línea roja), requieren aproximadamente el mismo tiempo. La primera población itera muchas veces, más en concreto, el doble de veces que la segunda población para finalizar la ejecución. Una ejecución del algoritmo sobre una misma población puede variar considerablemente dependiendo del número de centros, o la disposición de los mismos.

Las distancias entre individuos siguen presentes, pero esta vez, al tener una complejidad menor, no debería afectar tanto usar la distancia *Euclídea* o *Manhattan*. O eso es lo que parece a simple vista. Como se comprobó en la Figura 4.12, el número de iteraciones para llegar a la condición de finalización importa, y usar una distancia u otra va a influir en el tiempo de ejecución. El número de iteraciones varía dependiendo de qué distancia se use, pues la *Euclídea*, aunque su cálculo es más lento, tiene una mayor precisión, lo que le da una gran ventaja frente a la distancia *Manhattan*. Esta última, al no ser tan precisa, puede

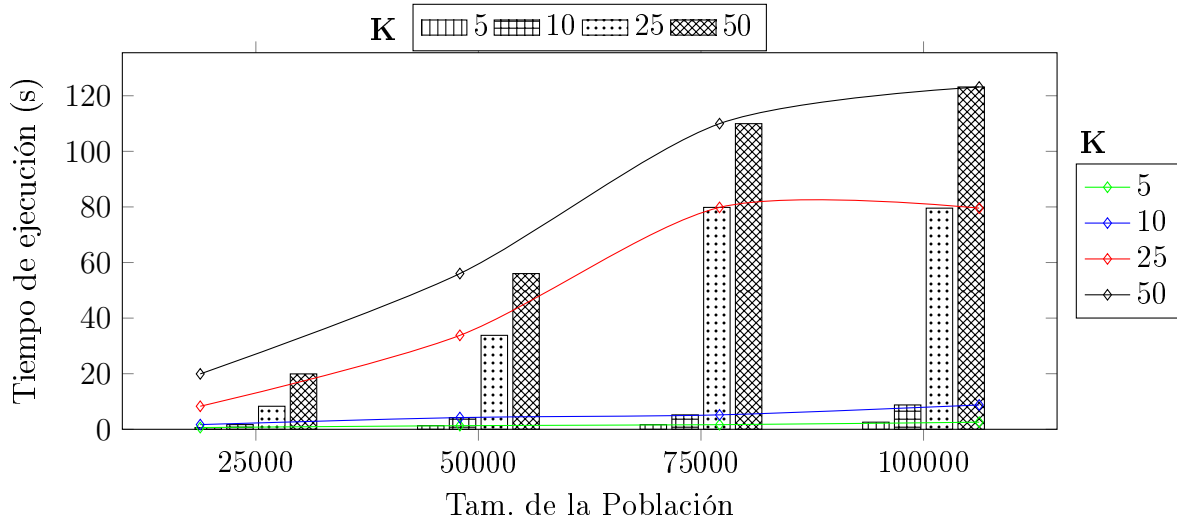


Figura 4.12: Variaciones en el número de *clusters* (K) en el algoritmo K-Medias

hacer que, aunque sea por poco, un individuo pertenezca a otro *cluster*, provocando una reacción en cadena que resulte en un aumento considerable en el número de iteraciones.

El estudio realizado para comprobar el rendimiento de la estrategia comentada en la Sección 3.2.2 con cinco procesos frente al algoritmo secuencial, se representa en la Figura 4.13, utilizando $K=10$ centros, y comparando también las distancias entre individuos (*Euclídea* y *Manhattan*). Los tamaños de las poblaciones utilizadas para medir estas pruebas se realizan como en las pruebas de las ordenaciones cuadráticas (ver Sección 4.2.1.1). Se puede apreciar que las funciones tienen picos, siendo más pronunciados en los algoritmos sin paralelizar. Como se comentó anteriormente, el tiempo de ejecución para una población puede variar dependiendo de la distancia implementada, además de la posibilidad de que una población con menor tamaño pueda tardar mucho más que una población mayor, debido a la disposición de los individuos y los *clusters* en la ejecución.

Comparando el algoritmo secuencial y el paralelizado se puede apreciar una mejora considerable, y debido a los picos, es interesante medir la evolución de los *speed-ups*. La Figura 4.14 muestra esta evolución, cuyos *speed-ups* son calculados con los tiempos utilizados en la anterior figura. Ambas distancias comienzan siendo volátiles, siendo algunas veces peor que el algoritmo secuencial ($speed-up < 1$) y otras veces superando por mucho el *speed-up*

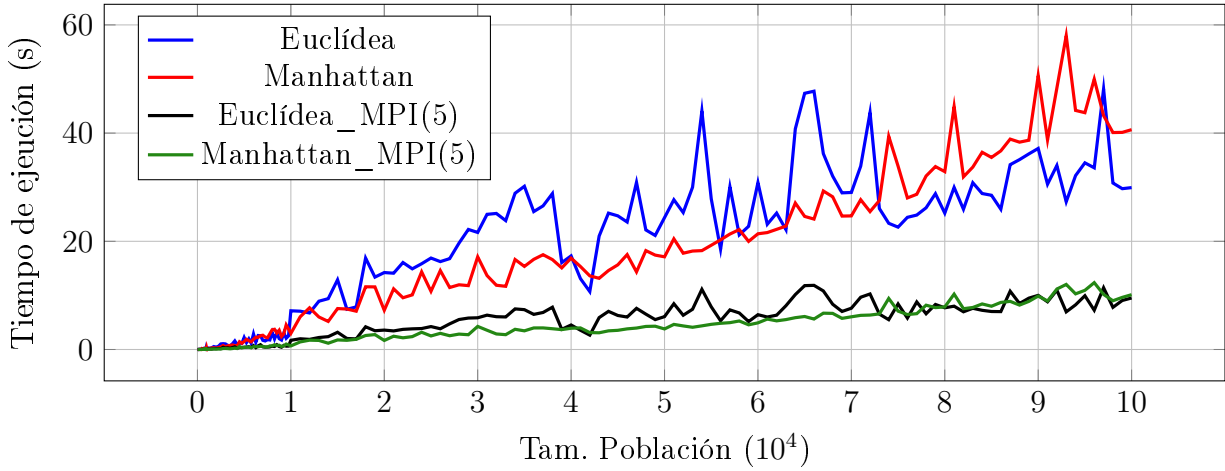


Figura 4.13: Tiempo de ejecución -con 5 procesos- de la primera estrategia del algoritmo K-Medias en ordenador de propósito general

ideal. A partir de diez mil individuos de población, el *speed-up* es equivalente al número de *workers* ejecutados. Tras analizar los resultados, observamos que, pese a que la distancia *Euclídea* es más precisa, a la larga es mejor aplicar distancia *Manhattan*, pues, aunque itere más veces, el coste es menor, llevando a conseguir mejores resultados. Se puede apreciar la línea azul superando en la mayoría de las veces a la línea roja, probando lo comentado.

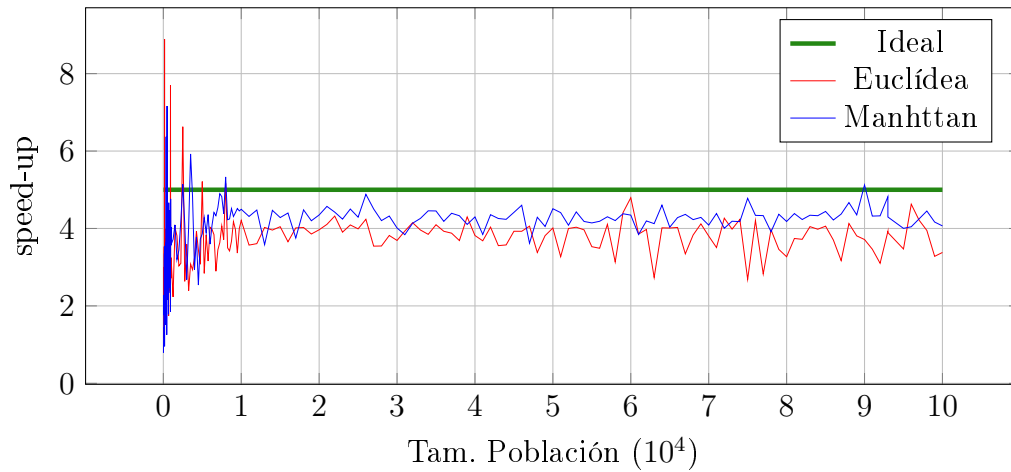


Figura 4.14: Speed-up de la primera estrategia del algoritmo K-Medias en ordenador de propósito general usando 5 procesos

Para este algoritmo, al contrario que el anterior, se pueden realizar pruebas con tamaños

de poblaciones mayores en el sistema distribuido. La siguiente prueba realizada comienza con una población de *20000* individuos, esta vez con cinco variables de entrada. Entre pruebas se aumenta ese mismo tamaño hasta llegar a *240000* individuos, utilizando en proporción una población seis veces mayor que en el ordenador de propósito general. Se usa el mismo valor de K ($K=10$), y se ejecuta la misma estrategia con *10*, *20*, *35*, *50*, *75*, *100* y *128* procesos. Como se muestra en la Figura 4.15, a partir de veinte procesos, la reducción del tiempo de ejecución se ralentiza. Con un número elevado de procesos, esta estrategia no consigue reducir el tiempo de ejecución en proporción a los procesos ejecutados, esto se debe a la gran cantidad de comunicaciones que se deben realizar para finalizar la ejecución.

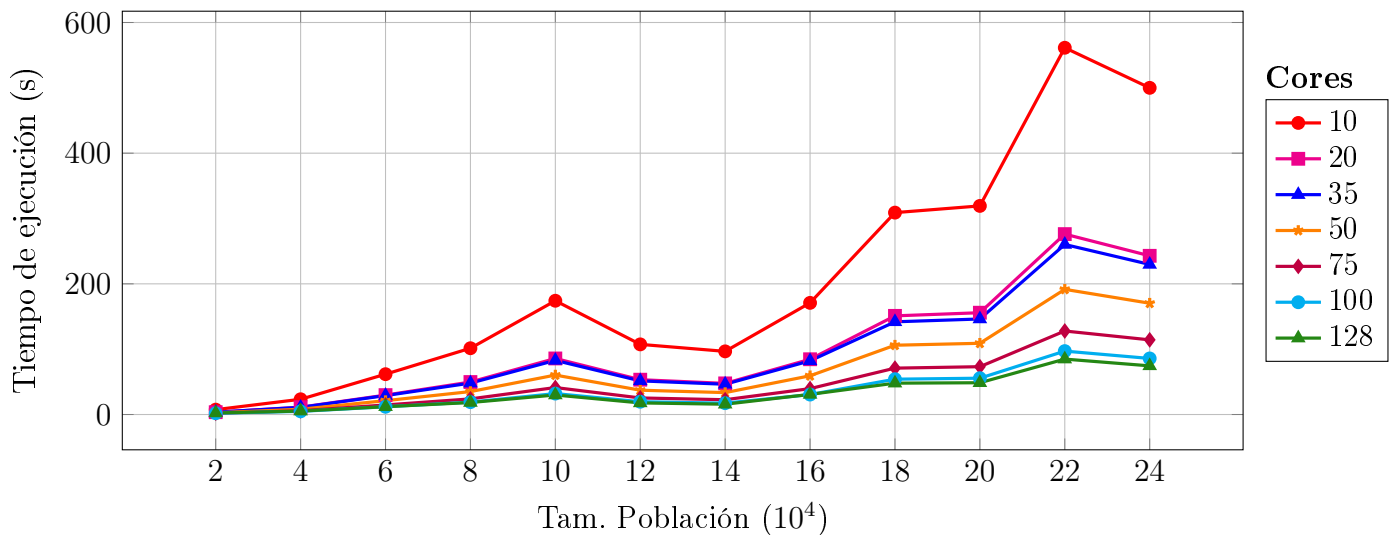


Figura 4.15: Tiempo de ejecución de la primera estrategia del algoritmo K-Medias en el Cluster

4.3.3. KNN

En cada iteración de este algoritmo de aprendizaje supervisado, se clasifica un individuo utilizando una población previamente categorizada. Al contrario que los algoritmos de aprendizaje no supervisado, que agrupan una población entera al finalizar la población. La complejidad temporal de este algoritmo es menor, y el valor de K no influye en el tiempo de ejecución como el algoritmo de *K-Medias*, pues al aumentar este valor solo aumenta el

número de los individuos más cercanos que se comprueban para categorizar el nuevo individuo. Este algoritmo usa dos poblaciones, y como se comentó en la Sección 3.2.3, las dos estrategias dividen una de las poblaciones para paralelizar el algoritmo.

Para las siguientes pruebas realizadas en el ordenador de propósito general, se fija la misma población utilizada en el algoritmo anterior, con un tamaño de 100000 individuos para la población a categorizar. La población inicialmente categorizada tiene un tamaño de mil individuos y se obtiene realizando una búsqueda exhaustiva con el algoritmo K-Medias. Esta búsqueda se realiza con valores de K en el intervalo de $[2, 20]$ centros, ejecutando, para cada uno, el algoritmo diez veces, calculando así la mejor agrupación. Se obtiene como resultado cuatro centros. Con estas dos poblaciones se ejecuta el algoritmo de *K-Vecinos más Cercanos* con un valor de $K=15$, un número impar para que no haya posibilidad de empates a la hora de asignar un *cluster* a cada individuo.

Primero comprobamos los dos métodos para el algoritmo secuencial, actualizar o no actualizar al categorizar un nuevo individuo. Si se actualiza la población conforme avanzan las iteraciones, la población final será mucho más precisa que si no se actualiza, pero el tiempo de ejecución aumentará considerablemente. La Figura 4.16 muestra los resultados. Si no se actualiza, la complejidad es lineal, pues la población categorizada se mantiene constante, y no se puede diferenciar cuál de las dos distancias ralentiza más la ejecución. Sin embargo, cuando se actualiza la población, se comprueba una vez más que la distancia *Euclídea* es más lenta que la *Manhattan*.

Después de comprobar el algoritmo secuencial pasamos a las estrategias para reducir el tiempo de ejecución. El algoritmo sin actualizar es rápido y es mejor estudiar el comportamiento con una población variable con el tiempo. Por eso se ejecutan las dos estrategias con cinco procesos. Podemos ver los resultados en la figura 4.17, con una reducción notoria en el tiempo de ejecución. Para la primera estrategia, dividir la población categorizada entre los *workers*, se realizan dos versiones, una en la que cada *worker* espera el individuo categorizado de la iteración anterior (línea de color verde), y otra en la que no se espera, sino que los

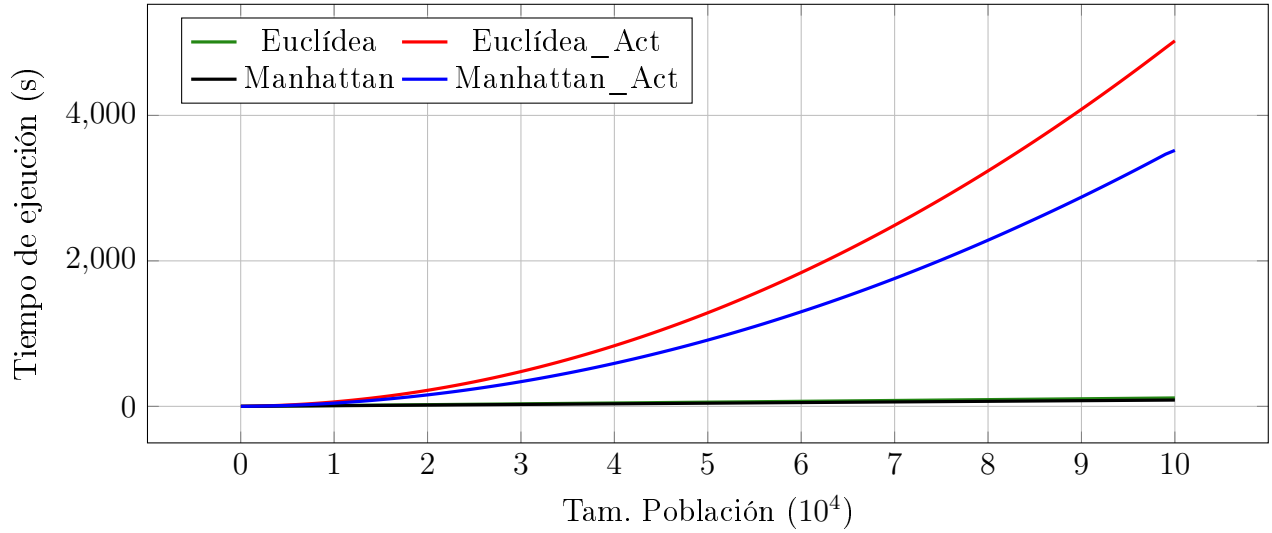


Figura 4.16: Tiempo de ejecución del algoritmo secuencial KNN en ordenador de propósito general

workers trabajan en la siguiente iteración mientras que el *master* agrupa el individuo (línea de color negro). La primera estrategia es ligeramente más rápida que la segunda (línea de color azul), y aun perdiendo tiempo esperando a la categorización del individuo (la primera versión), sigue finalizando antes que la segunda estrategia.

En cuestión de complejidad espacial la segunda estrategia consume mucha más memoria. Al finalizar la ejecución, cada *worker* tiene una copia entera de la población categorizada, mientras que en la primera mejora se divide esta población entre los procesos.

Comparando las evoluciones de los *speed-ups* en las estrategias, se puede concluir que al principio es mejor dividir la población a predecir, pero a largo plazo es más efectivo dividir la población categorizada, además de tener menos complejidad espacial.

En algoritmos pasados ya hemos visto el funcionamiento de varias estrategias con tamaños de poblaciones elevados. Esta vez, para las pruebas en el sistema distribuido, ejecutamos la misma prueba que antes, pero con más procesos en paralelo. Se ejecutan 10, 20, 35, 50, 75, 100 y 128 procesos sobre la mejor estrategia obtenida en el estudio anterior, comprobar el *speed-up* al usar muchos procesos. La Figura 4.19 muestra que, al aumentar los procesos, no se reduce considerablemente el tiempo de ejecución, generando sobrecarga a partir de veinte

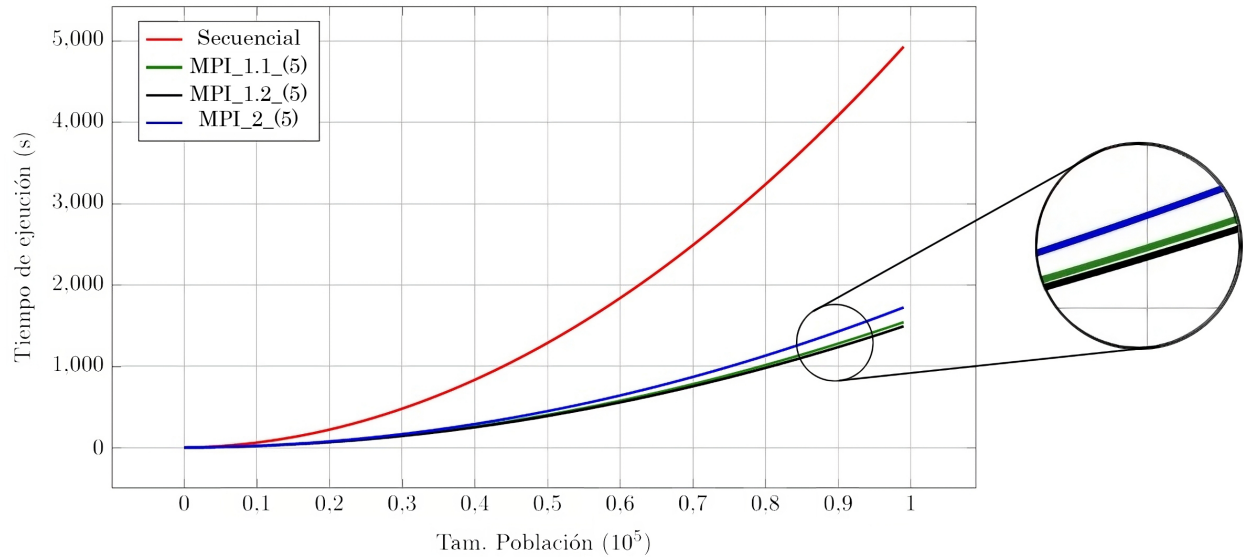


Figura 4.17: Tiempo de las estrategias del algoritmo KNN en ordenador de propósito general

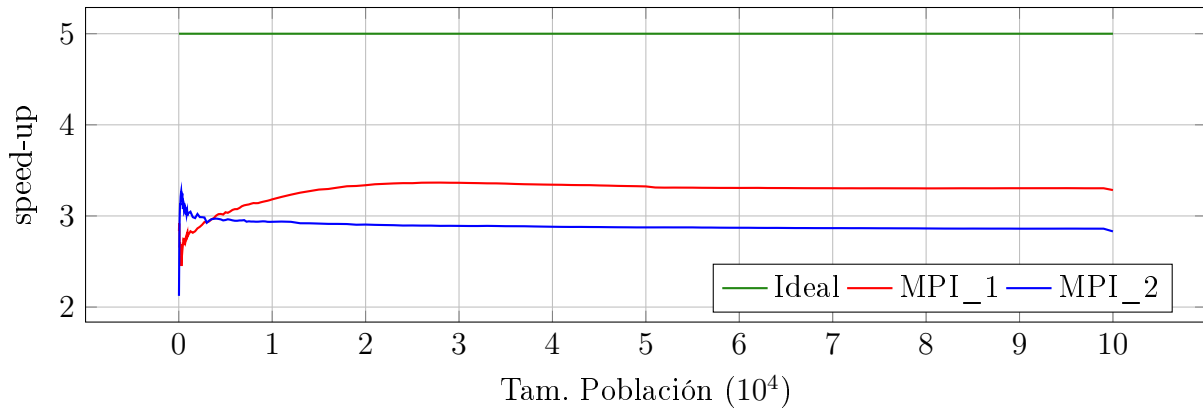


Figura 4.18: *Speed-ups* de las estrategias del algoritmo KNN en ordenador de propósito general

procesos. Al igual que en el algoritmo *K-Medias*, aumentar el número de procesos provoca que, aunque se reduce el tiempo de ejecución en cada iteración, el tiempo de comunicación (overhead) entre iteraciones aumenta.

4.4. Q-Learning

Para el aprendizaje por refuerzo, cuyos dos algoritmos se comentaron en la Sección 2.2, primero se estudia el algoritmo de *Q-Learning*. El otro algoritmo, *Deep Q-Network*, se basa

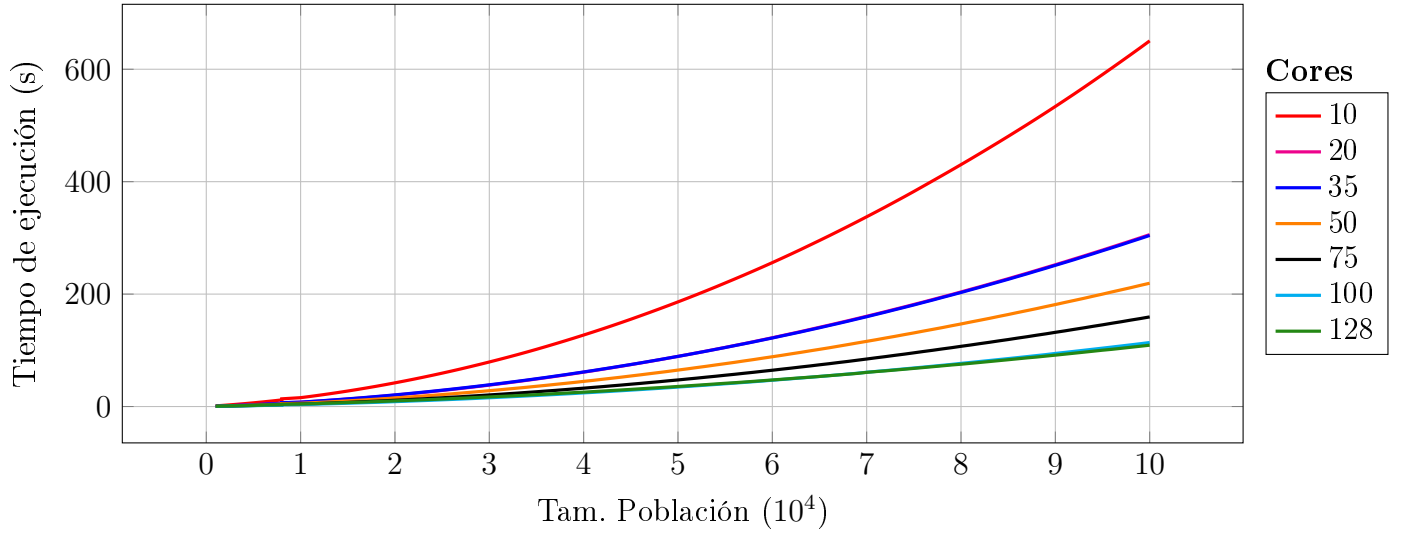


Figura 4.19: Tiempo de la primera estrategia del algoritmo KNN en Cluster

en redes neuronales, estudio que se realiza posteriormente en la Sección 4.6.

Antes de entrar en profundidad con las estrategias comentadas en la Sección 3.3, primero estudiamos el comportamiento del algoritmo de manera secuencial, con y sin preprocesado del entorno. Este preprocesado consiste en recorrer la matriz entera eliminando estados inaccesibles (el agente se sitúa en un muro) y acciones que no queremos que el agente ejecute, como chocar con una pared. Se ejecuta con tres laberintos diferentes, con 30, 50 y 100 filas. La Figura 4.20 muestra una leve reducción en el tiempo de ejecución. Además, obtiene mejores resultados con una mayor variedad de combinaciones de hiper-parámetros. Al reducir las acciones disponibles, el agente tiene una mayor probabilidad de explorar más el laberinto, generando más combinaciones con las cuales aprender el camino óptimo hasta la meta.

Una buena configuración de hiper-parámetros genera que el agente logre alcanzar su objetivo. En entornos de gran tamaño, algunas veces, es complicado encontrar configuraciones que funcionen, lo que provoca un aumento en el tiempo dedicado a la fase de entrenamiento para encontrar estas combinaciones. Por este motivo, se desarrolla una estrategia para encontrar combinaciones de los hiper-parámetros realizando una búsqueda exhaustiva.

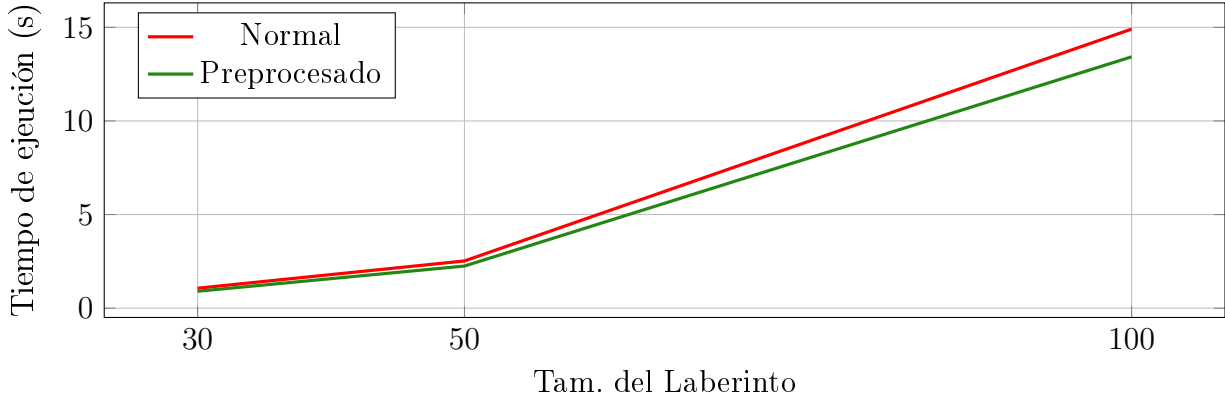


Figura 4.20: Tiempo de ejecución del algoritmo secuencial Aprendizaje por Refuerzo en ordenador de propósito general

Consiste en ejecutar en muchos procesos el algoritmo secuencial sin preprocesar con diferentes combinaciones. Se inicializa 100 filas y columnas, y con una precisión de 0.01 , es decir un 1% , el *master* aumenta los valores de los hiper-parámetros. Cuando uno de estos llega al 100% se reinicia y aumenta el siguiente, así hasta cubrir todas las posibles combinaciones. Al ser tres hiper-parámetros hay 10^6 combinaciones distintas. El *master* envía a cada *worker* diferentes combinaciones, y cuando terminan de procesar una combinación, envían de vuelta un mensaje de confirmación. Si termina con éxito, el *master* almacena en un fichero la combinación de hiper-parámetros, junto con los movimientos requeridos, así como las veces que el algoritmo falla y llega a la meta. En caso contrario no almacena nada. Dicha prueba se realizó en el sistema distribuido, con 128 procesos, y ha tardado siete días completos en finalizar la prueba, obteniendo 500 combinaciones distintas, de las cuales, solo la mitad dan con el camino óptimo.

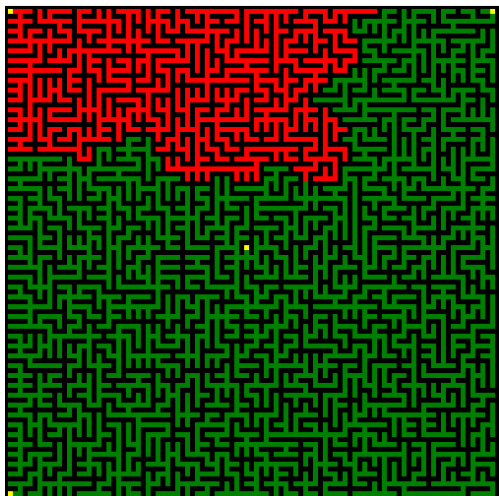
Una vez estudiado las combinaciones de hiper-parámetros, podemos pasar a las pruebas de las estrategias para reducir el tiempo de ejecución del algoritmo. Para ello, se usa un laberinto de cien filas y columnas, con una combinación de hiper-parámetros que sea óptima (alguna de las previamente encontradas). El algoritmo secuencial ejecuta 100 episodios (iteraciones que finalizan al llegar a la meta), mientras que la estrategia paralelizada, ejecuta, con cinco procesos, 400 episodios. Esta estrategia, usando el modelo *Master-Worker* se

basa en la misma idea de encontrar las combinaciones de hiper-parámetros, pues se ejecuta en paralelo el algoritmo secuencial, pero esta vez en diferentes posiciones. Al final de la ejecución, el *master* realiza la media de las experiencias obtenidas en todo el entorno.

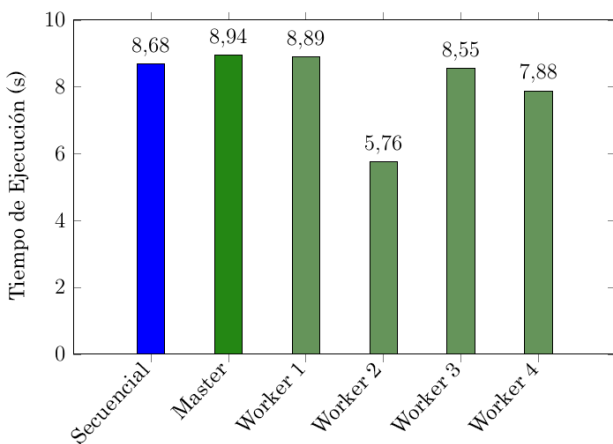
La imagen a la izquierda de la Figura 4.21 muestra el mapa de calor del algoritmo secuencial y la estrategia. En rojo se representan las celdas que más veces han sido visitadas por el algoritmo secuencial, mientras que en verde las del paralelizado. En amarillo se representan los puntos de salida, siendo el superior izquierdo el del algoritmo secuencial y un proceso *worker* de la estrategia paralelizada (para la estrategia paralelizada hay que garantizar que haya al menos un proceso en el punto origen). Para obtener este mapa se hacen unas comparaciones de las celdas visitadas, haciendo la media en el caso de la estrategia, pues sin hacerla se obtendría una mapa completamente verde. Se puede observar que el algoritmo secuencial visita más veces la parte superior izquierda, debido a que la media de veces que se visita esa zona en la estrategia se divide entre cuatro, y al ser menos probable que todos los *workers* la visiten, hace que la ejecución secuencial la visite más veces, ya que siempre se inicializa por esa zona.

Los tiempos de ejecución son similares. El tiempo de ejecución de la estrategia paralelizada viene dada por el mayor tiempo de ejecución de los *workers*. El *worker* que más tiempo consume es el que está posicionado en la misma salida que el algoritmo secuencial, por lo que por la aleatoriedad y el tiempo de comunicación con el proceso *master* provoca que la ejecución total de la estrategia sea más lenta. No obstante, esta estrategia visita muchas más celdas en el laberinto, además de ejecutar cuatro veces más episodios sumando los ejecutados en cada proceso.

La segunda estrategia, dividir el entorno entre los procesos ejecutados, no funciona correctamente. En la fase de entrenamiento no se ha conseguido finalizar, en ninguna de las pruebas realizadas, en una matriz de treinta filas y columnas (el laberinto de menor tamaño generado previamente). No obstante, al ejecutar la estrategia con valores aprendidos en un entrenamiento secuencial, sí se logra llegar a la meta. Hemos comprobado con varios



(a) Mapa de calor del laberinto 100×100



(b) Procesos y modo de ejecución

Figura 4.21: Tiempo de ejecución y mapa de calor de la primera estrategia del algoritmo Aprendizaje por Refuerzo en ordenador de propósito general

hiper-parámetros y ninguno termina el entrenamiento. La teoría más razonable es que esta estrategia está muy influenciada por la combinación de hiper-parámetros, provocando que una mala configuración lleve a un mal aprendizaje, generando bucles infinitos.

4.5. Algoritmos Evolutivos

Este algoritmo no se ejecuta sobre unos datos previamente generados, sino que en cada ejecución crea desde cero los individuos que van a evolucionar para optimizar una función de evaluación. Las pruebas realizadas en esta sección en el ordenador de propósito general se ejecutan sobre los siguientes tamaños de poblaciones $[25, 50, 100, 200, 500, 1000, 2000]$ con 100 generaciones. Para garantizar la supervivencia de los mejores individuos se utiliza un 5 % de elitismo.

Como cada individuo tiene su propia estructura, no se puede aplicar los mismos métodos para las partes del algoritmo. El cruce y la mutación varían dependiendo del individuo, pero se eligen los más básicos. La selección es común para todos los individuos. Debido a que solo juega con los valores *fitness* (aptitud del individuo para resolver el problema) calculados en

las funciones de evaluación, que se comentaron en la Sección 3.4. Esta selección consiste en elegir aleatoriamente los individuos, cuyas probabilidades de selección aumentan o disminuyen dependiendo de su valor de *fitness*. Para la función de evaluación, además de contar con el cálculo del *fitness*, también cuenta con otras funcionalidades como el desplazamiento de individuos, que consiste en garantizar que todos tengan un valor *fitness* positivo, además de controlar la diversidad con un escalado lineal. Para los individuos representados como árboles, se añade un control de *bloating*, controlando así la altura de los individuos, generando un leve aumento en el tiempo de la función de evaluación, pues si un individuo supera un límite de altura en el árbol, se sustituye con uno nuevo generado de manera aleatoria. Las complejidades del cálculo *fitness* y las variables que afectan a los tamaños de las pruebas de cada individuo se determinan de la siguiente forma:

1. Binario. Función matemática. Tiene un coste lineal $O(N)$, siendo N el número de bits, pues solo tiene que convertir de binario a real y aplicar la función matemática. El tamaño de las pruebas es condicionado por la precisión, variable que refleja el grado de exactitud necesario de la codificación binaria para representar un valor numérico real. Cuanto mayor sea ésta, más bits se necesitan. Con una precisión de 2 decimales se necesitan 11, y con una precisión de 10 son necesarios 38 bits. Como el problema se mide en un espacio bidimensional, se necesitan dos números reales, sumando en total 22 y 76 bits respectivamente.
2. Real. El utilizado en el problema del aeropuerto. Tiene un coste cuadrático $O(N*M)$ siendo N el número de aviones, y M el de aeropuertos, aunque el coste amortizado acaba siendo lineal, por tener un número de aviones mucho mayor al de pistas. El tamaño de las pruebas para este problema es condicionado por la complejidad temporal, por lo que el número de aviones y aeropuertos son las variables que se van a estudiar.
3. Árbol. El utilizado en el problema del cortacésped. Tiene un coste lineal $O(N)$, siendo N el número de acciones (*ticks*) a disposición del agente para intentar cortar el máximo

número de celdas. El número de *ticks* es la variable que modifica el tiempo ejecución. El tamaño del entorno (la matriz cuadrada que representa el jardín) no aumenta el tiempo de ejecución, pues tarda el mismo tiempo en ejecutar X ticks en cualquier matriz, solo aumenta la complejidad espacial de la función de evaluación.

La Tabla 4.1, muestra los identificadores, con los valores de las variables, anteriormente mencionadas. En la parte izquierda los individuos binarios, en la central los reales y a la derecha los árboles.

Identificador	Precisión	Identificador	Aviones	Pistas	Identificador	Ticks
P2	2	AER1	12	3	M10x10	150
P10	10	AER2	25	5	M100X100	1500
		AER3	100	10		

Cuadro 4.1: Variables de cada individuo utilizadas en los experimentos

Primero, veamos cómo afecta a cada individuo modificar las variables de cada problema. La Figura 4.22 muestra dichos tiempos. La primera gráfica pertenece a los individuos binarios, y se puede apreciar que modificar la precisión no supone un coste elevado. Aumentar por cinco la precisión de la representación binaria ($precisión(P10)/precisión(P2) = 5$) supone un aumento en el tamaño del array de bits de 3.45 veces ($76/22 = 3.45$) su tamaño. Sin embargo, el tiempo de ejecución no es proporcional al aumento del tamaño de bits, sino que es aproximadamente dos veces más lento. La gráfica del medio, representando a los individuos reales, tiene una mayor complejidad, por eso se puede apreciar una diferencia significativa al utilizar la segunda especificación (*AER2*) con respecto a la tercera (*AER3*), siendo seis veces más lenta. La última gráfica, muestra los resultados del problema de los individuos representados como árboles. Se puede observar que la primera función (línea azul) es cuatro veces más rápida que la segunda (línea roja). El tiempo de ejecución no es proporcional al número de *ticks* ejecutados, pues el segundo problema ejecuta diez veces más *ticks* que el primero. Las gráficas se muestran de forma lineal porque el número de generaciones es estático, es decir, solo aumenta el tamaño de la población. De otro modo sería exponencial y el tiempo de ejecución se elevaría.

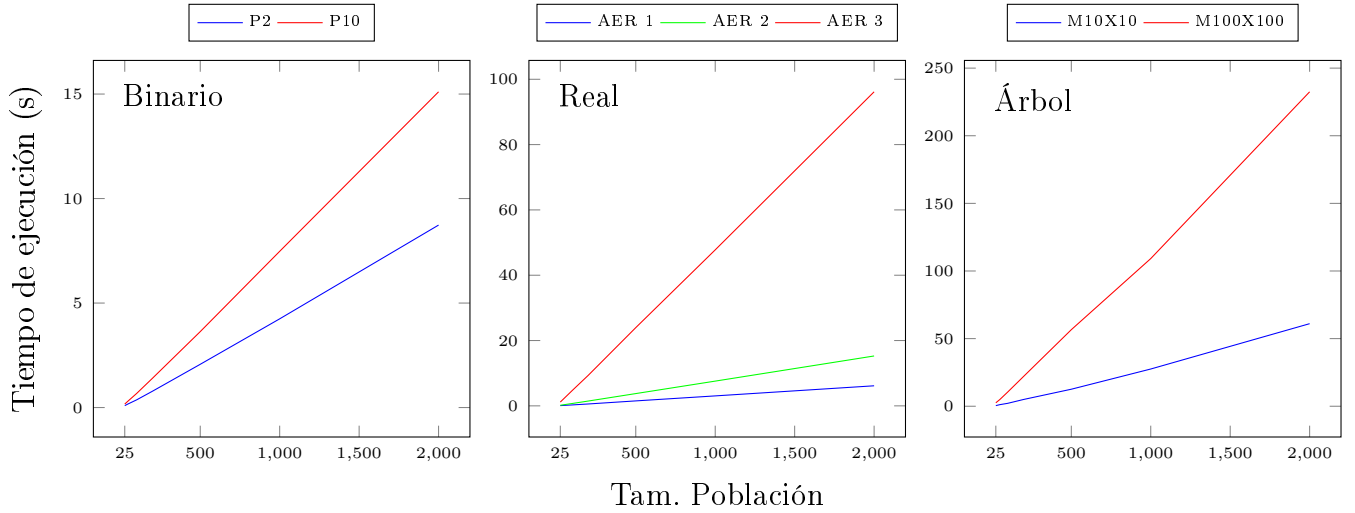


Figura 4.22: Tiempos de ejecución de los algoritmos evolutivos secuenciales

Antes de realizar las pruebas de las tres estrategias comentadas en la Sección 3.4, vamos a comentar los tiempos de ejecución de las partes del algoritmo evolutivo, inicialización, evaluación, selección, cruce y mutación. Cada prueba mide el tiempo de ejecución para un individuo (dos individuos en el caso del cruce). La Tabla 4.2, muestra los resultados obtenidos, resaltando en rojo aquellos métodos que más tiempo consumen. Los resultados de los individuos binarios muestran que el cruce y la mutación son las partes que más tiempo consumen (inicialización no cuenta, pues solo se ejecuta una vez). La evaluación de estos individuos recorre los bits para convertirlo a un número real, pero no cambian los valores, reduciendo así el tiempo de ejecución. Los individuos reales y árboles tienen una función de evaluación con mayor complejidad, lo que provoca que esta parte sea la que más tarde.

Cabe destacar que estos tiempos no son la única medida que se necesita para las estrategias. El tiempo de comunicación entre procesos va a aumentar considerablemente el tiempo de ejecución, siendo, el individuo binario, el tipo de individuo que más desventaja tiene, pues al tener más datos consumirá más tiempo al enviar y recibir mensajes.

Empezando con las estrategias de la Sección 3.4, *modelo de islas* consiste en dividir los individuos de la población entre “islas” (procesos). La configuración de las islas no influye en la reducción del tiempo de ejecución, en una configuración u otra solo varía como se

Datos	Funciones	Init(1)	Evaluación(1)	Selección(1)	Cruce(2)	Mutación(1)
Precision: 2	Binario	2.56e-05	4.4e-06	8.56e-06	1.36e-05	1.53e-05
Precision: 10	Binario	3.33e-05	5.44e-06	8.9e-06	1.71e-05	1.88e-05
aviones: 12 pistas: 3	Aeropuerto 1	7.04e-06	2.55e-05	4.12e-06	1.48e-05	2.764e-06
aviones: 25 pistas: 5	Aeropuerto 2	1.36e-05	6.55e-05	4.62e-06	2.4e-05	3.43e-06
aviones: 100 pistas: 10	Aeropuerto 3	3.97e-05	4.3e-04	8.05e-06	4.18e-05	1.04e-05
M10x10 ticks: 150	Árbol	6.12e-05	6.47e-05	7.92e-05	2.33e-05	3.47e-07
M25x25 ticks: 400	Árbol	6.16e-05	1.65e-04	7.88e-05	2.32e-05	3.7e-07
M100x100 ticks: 800	Árbol	6.41e-05	3.66e-04	8.07e-05	2.09e-05	3.23e-07

Cuadro 4.2: Tiempos unitarios de las partes del algoritmo evolutivo para cada individuo

garantiza la supervivencia de los mejores individuos en la población general. Si usamos la topología en estrella, hay que reservar un proceso para que actúe como *master* para que éste realice el proceso de comunicación cada X generaciones. Las demás topologías (red y anillo) no tienen un proceso *master* por lo que se optimiza de mejor forma los recursos computacionales. Las pruebas realizadas a continuación se han ejecutado con la topología de anillo, con cuatro procesos. La Figura 4.23 muestra los resultados en forma de malla de 2×2 con los tiempos de ejecución de los algoritmos evolutivos con esta estrategia, separando las pruebas de los individuos reales (segunda fila) debido a la diferencia de tiempos con respecto al problema con mayor tamaño (*AER3*). Como se puede apreciar, se logra obtener una reducción del tiempo de ejecución proporcional al número de procesos ejecutados. El estudio de los *speed-ups* (figura 4.24) para los problemas de mayor tamaño de cada individuo, confirma la proporcionalidad de la reducción del tiempo de ejecución con respecto al número de procesos ejecutados. La primera gráfica de la primera fila (ver Figura 4.23) muestra los resultados obtenidos para los individuos binarios, consiguiendo reducir el tiempo de ejecución. La segunda gráfica de esta misma fila muestra los resultados para los árboles, cuyo tiempo de ejecución de la estrategia sobre el segundo problema (*1500 ticks*) es aproximadamente igual a los obtenidos para la ejecución secuencial del primer problema (*150 ticks*). Como se

usan cuatro procesos y el primer problema es cuatro veces más rápido que el segundo, los resultados de estas ejecuciones se solapan, provocando que la línea verde y azul coincidan. Las gráficas de los individuos reales de la segunda fila, muestran un mismo comportamiento que las gráficas anteriores con respecto a la reducción del tiempo de ejecución.

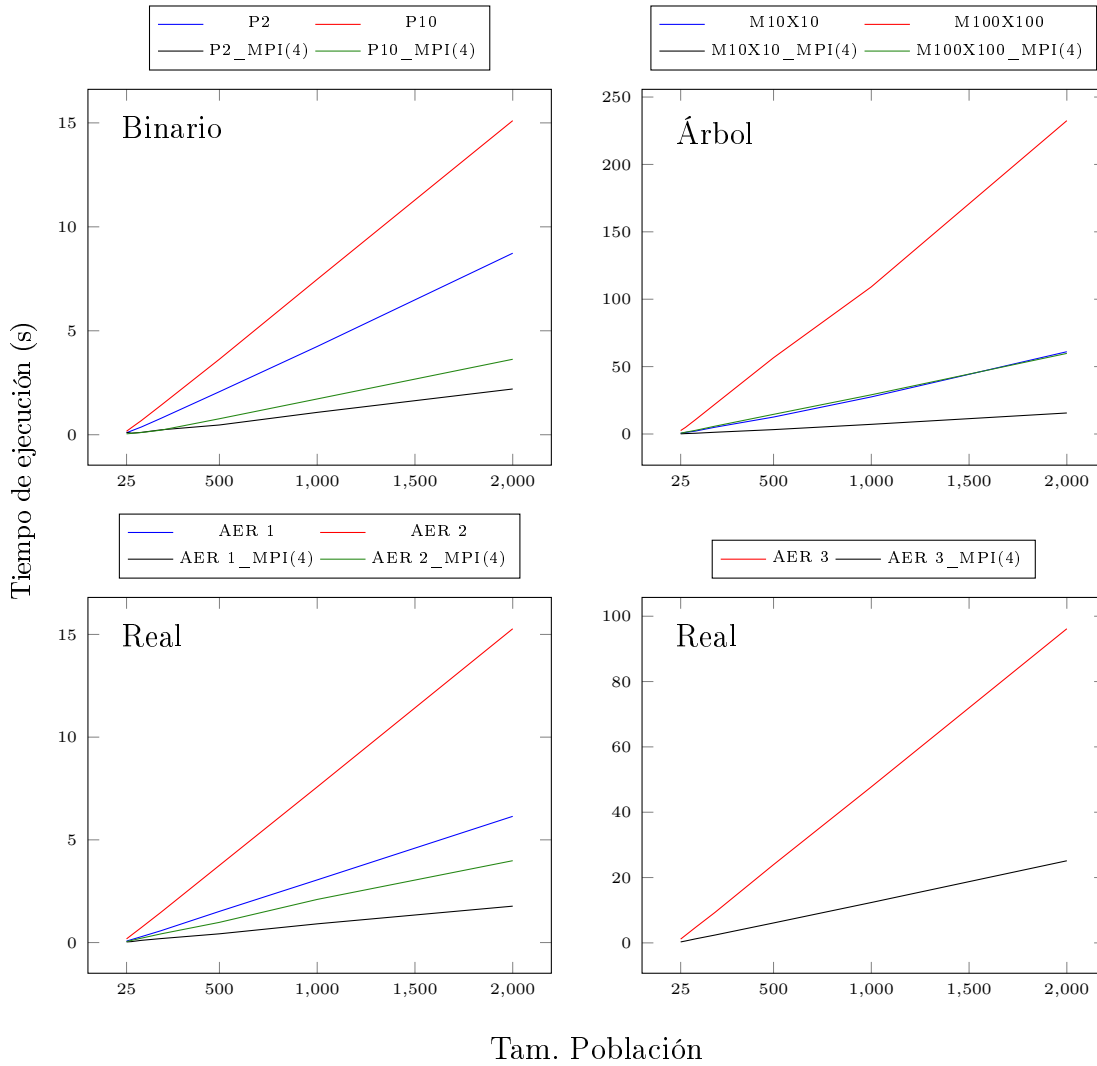


Figura 4.23: Tiempos de ejecución de la estrategia *modelo de islas* de los algoritmos evolutivos en ordenador de propósito general

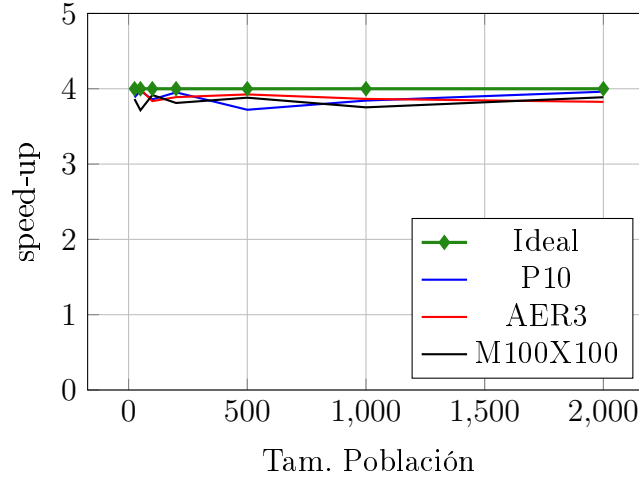


Figura 4.24: *Speed-ups* de la estrategia *modelo de islas* de los algoritmos evolutivos en ordenador de propósito general

La estrategia de *dividir la población* entre los procesos tiene una complejidad mayor en lo que a lógica de programación se trata. Con el modelo de comunicación *Master-Worker* y una población de individuos, el *master* se encarga de dividir y enviar a los *workers* la población sobre la cual tienen que ejecutar las partes del algoritmo: cruce, mutación y evaluación en cada generación. Las siguientes pruebas se ejecutan con cuatro *workers*, cinco procesos en total contando al *master*. La Figura 4.25 muestra, en forma de malla 2×2 , los resultados obtenidos para los tres diferentes individuos.

La primera gráfica de la primera fila muestra los resultados de los individuos binarios, unos tiempos de ejecución muy parecidos. Para el primer problema, usando 22 bits, no se logra reducir el tiempo de ejecución, pues se puede ver que empiezan de forma similar, pero con mil individuos se empiezan a distanciar. Esto no ocurre con el segundo problema (77 bits por individuo), la estrategia logra reducir levemente el tiempo de ejecución, y con dos mil individuos, la ejecución sigue siendo un poco más rápida. El factor que frena a esta estrategia de reducir el tiempo de ejecución es la complejidad de las operaciones a realizar en cada parte del algoritmo, pues son muy simples. Además, la comunicación entre procesos al enviar y recibir muchos bits aumenta el tiempo de ejecución. La segunda gráfica de la misma fila (ver Figura 4.25) muestra los resultados de los individuos representados como árboles,

siendo resultados parecidos a la anterior gráfica comentada. Para el primer problema (*150 ticks*) no se consigue reducir el tiempo de ejecución, no obstante, para el segundo si se logra, obteniendo, para la última población (*2000* individuos), un *speed-up* de *1.84*. Las pruebas en los individuos reales, al igual que para la estrategia anterior, se dividen en dos gráficas para poder ver con mayor exactitud los resultados obtenidos, estas gráficas se sitúan en la segunda fila (ver Figura 4.25). Al contrario que los otros dos individuos, este individuo si alcanza una reducción del tiempo de ejecución con todos los tamaños de problemas. La gráfica de la izquierda muestra que la estrategia en los dos primeros tamaños alcanza un buen rendimiento, pero el tercer problema, al ser más grande, tiene una reducción del tiempo de ejecución más notoria. Como muestra la gráfica de la derecha, se puede alcanzar un *speed-up* de *2.81*.

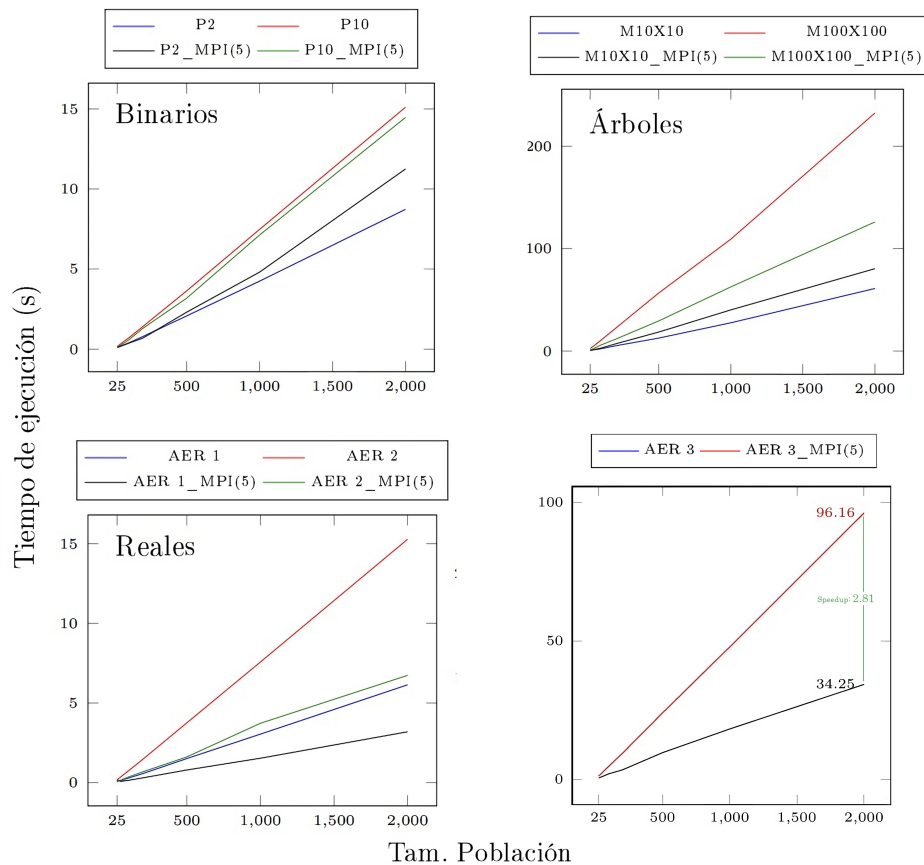


Figura 4.25: Tiempo de ejecución de la estrategia *dividir población* de los algoritmos evolutivos de en ordenador de propósito general

La estrategia *pipeline* mezcla el modelo *Master-Worker* con segmentación. El proceso *master* se encarga de generar una población dividida entre N (número de *workers*) subpoblaciones que envía al siguiente proceso (primer *worker*). Cuando genera todas las subpoblaciones, se queda en un estado de recepción de mejores individuos. Cada proceso envía a su siguiente los datos procesados según su tarea, generando un flujo constante de trabajo. Esta estrategia varía para cada individuo, debido a que los tiempos en cada parte del algoritmo son distintos para cada uno. Estos tiempos se estudiaron previamente en la Tabla 4.2. La primera prueba se realiza sobre los individuos binarios, con *precision*=10, cuyos procesos ejecutados se estructuran de la siguiente forma:

- Con cuatro procesos el *master* se encarga de inicializar. Los *workers* se dividen en tres pipes; el primer pipe se encarga de la evaluación y selección, el segundo del cruce y el tercero de la mutación.
- Con siete procesos: se duplica la ayuda para los *workers* en cada pipe.

Los resultados son plasmados en la Figura 4.26, logrando reducir satisfactoriamente el tiempo de ejecución. El funcionamiento de *pipeline* reduce el tiempo de paso de mensajes, al optimizar la paralelización de las partes del algoritmo.

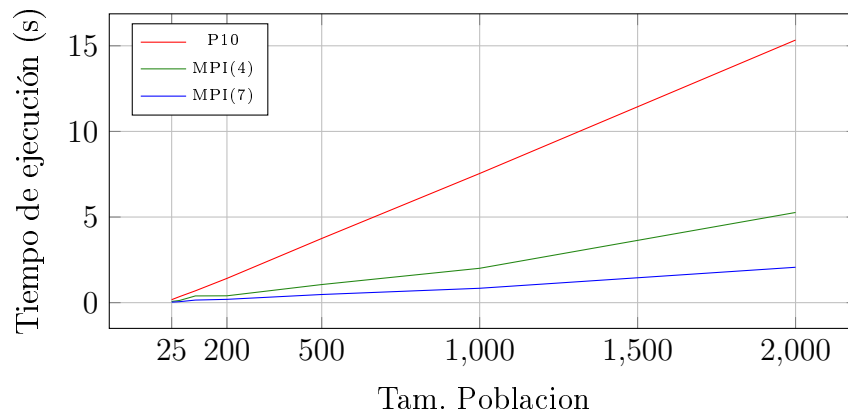


Figura 4.26: Tiempo de ejecución de la estrategia *pipeline* en los individuos binarios del algoritmo evolutivo en ordenador de propósito general

Los individuos reales y árboles tienen tiempos de ejecución muy parecidos. Es por eso que se obtendrían los mismos resultados al aplicar la misma repartición de tareas, siendo esta la siguiente:

- Con seis procesos: el *master* se encarga de inicializar. Los *workers* con *ids* en el intervalo $[1-4]$ se encargan de la evaluación, pues esta parte del algoritmo consume cuatro veces más tiempo que los restantes. El último worker se encarga de la selección, cruce y mutación.
- Con diez procesos: se duplica la ayuda para los *workers* en la función de evaluación. Alcanzando, con este reparto de tareas, una igualdad en los tiempos de ejecución de los dos tipos de procesos *worker*. Es decir, con ocho *workers* se logra reducir el tiempo de ejecución al mismo tiempo que el del *worker* que realiza las otras partes del algoritmo.

Como muestra la Figura 4.27, los individuos reales también presentan un buen rendimiento. Aunque duplicando los procesos *workers* de la función de evaluación (línea azul), se obtiene unos tiempos de ejecución similares a los obtenidos con seis procesos.

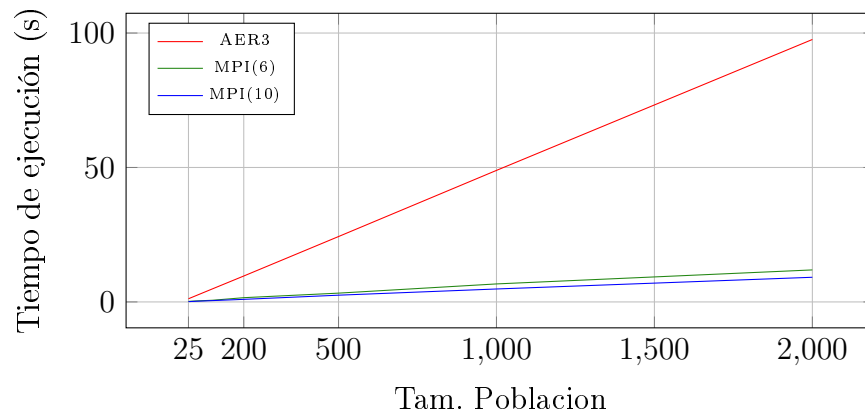


Figura 4.27: Tiempo de ejecución de la estrategia pipeline en los individuos reales del algoritmo evolutivo en ordenador de propósito general

En el *cluster* se han realizado pruebas para las estrategias de *dividir la población* y *pipeline*. La estrategia de *modelo de islas* no se ha realizado debido a que ya se han realizado varias pruebas en el sistema distribuido con esta estructura de dividir el algoritmo secuencial

entre varios procesos. Ambas pruebas tienen los siguientes tamaños de poblaciones $[1000, 2000, 5000, 7000]$. La estrategia de *dividir la población* se ejecuta sobre individuos reales, con 10, 20, 50, y 100 procesos. La Figura 4.28 muestra dichos resultados. Como se puede ver en el gráfico, a partir de 20 procesos la reducción del tiempo de ejecución empieza a ralentizarse. En el último tamaño de poblaciones, se logra obtener un menor tiempo con 50 procesos que al usar 100. La sobrecarga producida por la comunicación entre un número elevado de procesos *workers* y el *master* generan dichos resultados.

La estrategia de *pipeline* se ejecuta sobre individuos árboles, siguiendo el mismo reparto de tareas que la utilizada para los individuos reales. Como se comentó antes, al llegar a 10 procesos se alcanza una igualdad en los tiempos de ejecución. Es por esos que los procesos ejecutados para esta prueba son múltiplos de este número, siendo 10, 20, 40 y 80. Entre cada prueba se duplican los procesos involucrados en cada tarea, incluyendo la inicialización de los individuos realizada por el proceso *master*. Los resultados obtenidos son mostrados en la Figura 4.29. Se puede apreciar un comportamiento similar a la Figura 4.28 del párrafo anterior, en el cual al llegar a 40 procesos se obtienen aproximadamente los mismos resultados que duplicando los procesos.

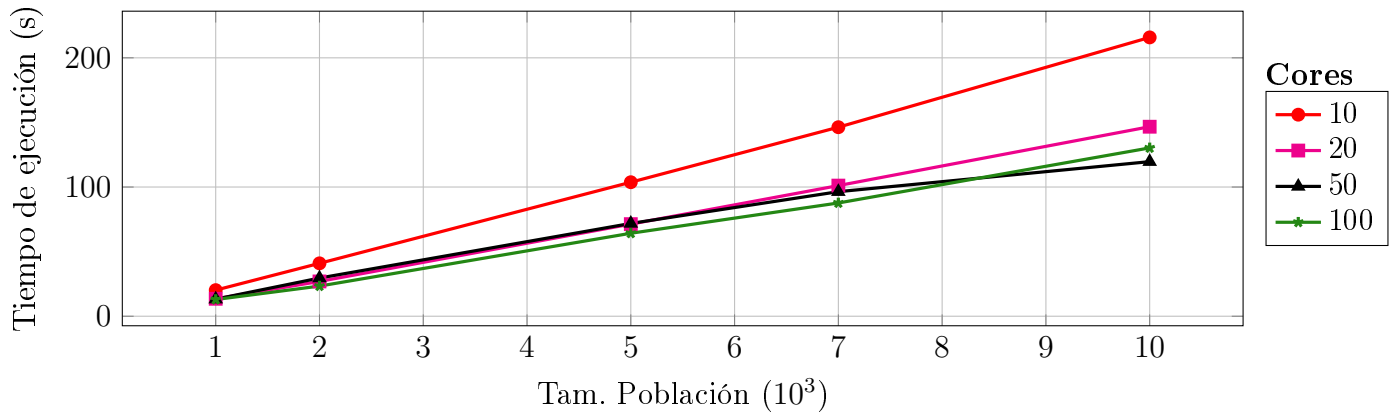


Figura 4.28: Tiempos de ejecución de la estrategia *dividir la población* en los individuos reales del algoritmo evolutivo en Cluster

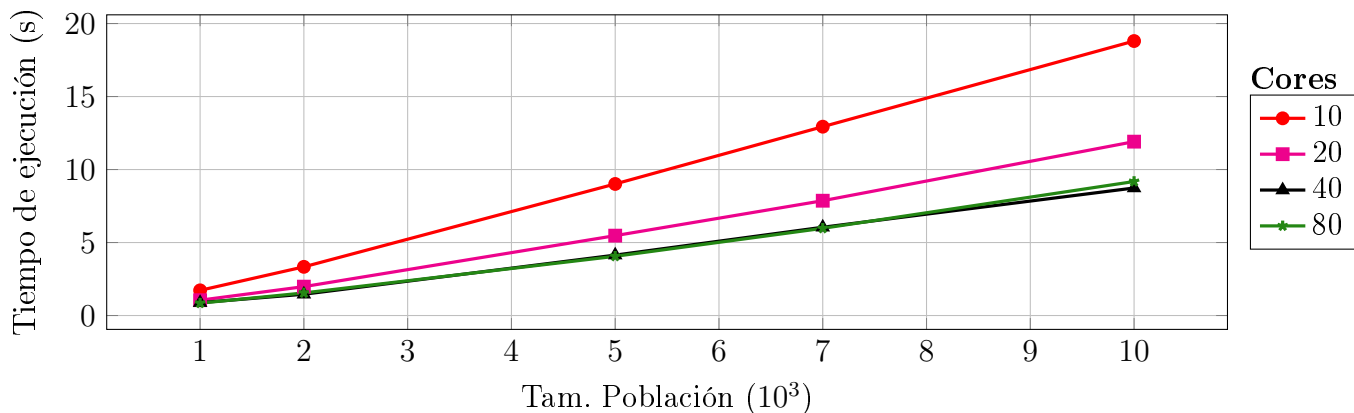


Figura 4.29: Tiempo de ejecución de la estrategia *pipeline* en los individuos árboles del algoritmo evolutivo en Cluster

4.6. Redes Neuronales

Este modelo de inteligencia artificial necesita una cantidad elevada de datos, utilizados en la etapa de entrenamiento para, de manera correcta, predecir los individuos. El algoritmo de DQN, comentado en la Sección 2.2.2, no necesita de un conjunto de datos, al ser un entorno en el cual un agente ejecuta acciones. Su etapa de entrenamiento consiste en ejecutar muchas veces diferentes ejecuciones para que aprenda a moverse por el entorno, modificando la red neuronal. Sin embargo, se pueden cambiar los estados con los cuales el agente comienza cada iteración, cambiando los variables del entorno para que sean accesibles. Ahora bien, para la predicción del índice de masa corporal (IMC) de un individuo, se necesita de una población con la cual enseñar a la red neuronal a predecir. Es por eso que se generan individuos de manera secuencial, variando sus valores para que no sean idénticos, y lograr así una población con la cual poder ejecutar el entrenamiento. Es importante resaltar que estos individuos tendrán una conexión con la realidad. Los individuos tienen alturas dado el siguiente intervalo en centímetros $[150, 200]$, y el peso varía con valores entre de 25 kilogramos por encima y debajo del peso ideal para cada altura ($IMC = 22.5$). Esto quiere decir que, si un individuo mide 180 centímetros, su peso se genera aleatoriamente con el siguiente intervalo de kilogramos $[55, 105]$.

La primera estrategia, *pipeline* de individuos, se logra generando en la capa de salida los individuos, siendo gestionados por el proceso *master*. Los demás procesos (los *workers*) gestionan las posteriores capas creadas. La siguiente prueba tiene una población de 2000 individuos, previamente generados como se comentó en el párrafo anterior. La red neuronal tiene una parte oculta con dos capas y cincuenta neuronas cada una (2×50). Con cinco repeticiones, se entrena la red neuronal con 10000 individuos en total, dando los resultados que se muestran en la Figura 4.30. Esta estrategia, tanto aplicando mensajes síncronos como asíncronos, no surte mucho efecto, pues en vez de reducir el tiempo de ejecución lo aumenta. En programación evolutiva, el flujo de mensajes es unidireccional, y no se pierde tanto tiempo entre mensajes. Este algoritmo, al tener dos métodos en diferentes direcciones, provoca un flujo bidireccional, y la comunicación entre procesos se ralentiza. Usando mensajes asíncronos, permite a cada proceso ejecutar antes el cálculo de *forward* (hacia adelante) y cuando recibe los errores los actualiza. Reduce muy poco el tiempo comparándolo con la versión síncrona. Además, hay que tener en cuenta que el flujo de mensajes hace que el modelo aprenda con valores desactualizados. Dependiendo de la población, puede situarse en un bucle en el cual aumenta y reduce los pesos, provocando un entrenamiento erróneo.

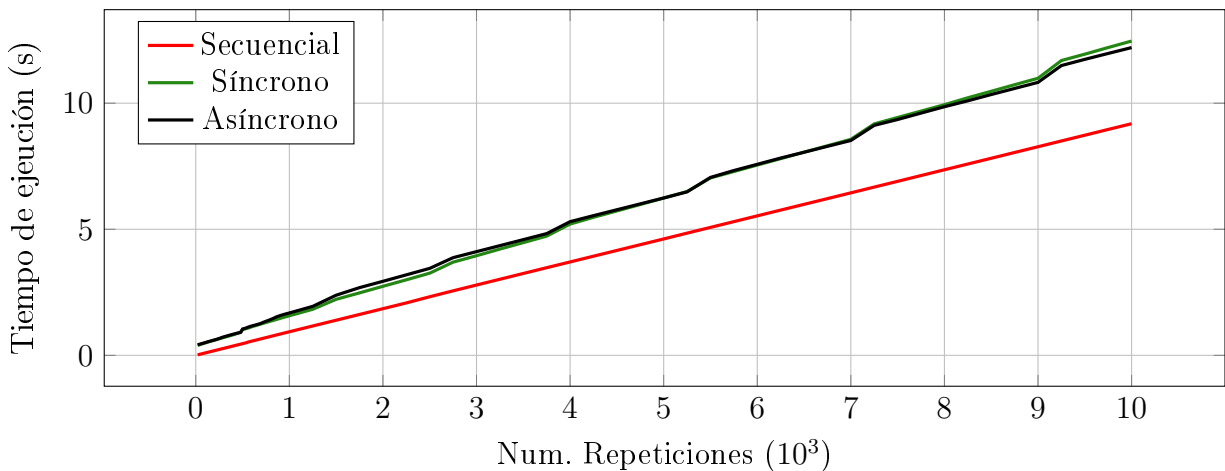


Figura 4.30: Tiempo de ejecución de la estrategia pipeline de Red Neuronal en ordenador de propósito general

La estrategia de dividir el proceso en entrenamiento entre varios procesos, ya se ha

comprobado que funciona correctamente en otros algoritmos como pueden ser *Q-Learning* y programación evolutiva con el modelo de islas, además de basarse ligeramente en la idea de *fine-tuning*. Esta vez, hay que tener en cuenta que la etapa de entrenamiento es un proceso iterativo en el cual se predice un individuo y se actualiza los errores cometidos, siendo un proceso complicado de lograr satisfactoriamente. La Figura 4.31 muestra la prueba realizada con una población de 80 individuos y 1000 repeticiones, sumando un total de 80000 individuos predichos en el entrenamiento. Se aplica el modelo *Master-Worker* para paralelizar el entrenamiento con 3 y 5 procesos, y una vez terminado enviar los pesos al *master* para realizar la media, intentando maximizar las predicciones finales. El *master* se encarga de dividir la población, siguiendo alguno de los métodos comentados en el final de la Sección 3.5. Se puede apreciar una reducción del tiempo de ejecución proporcional al número de procesos *worker* ejecutados.

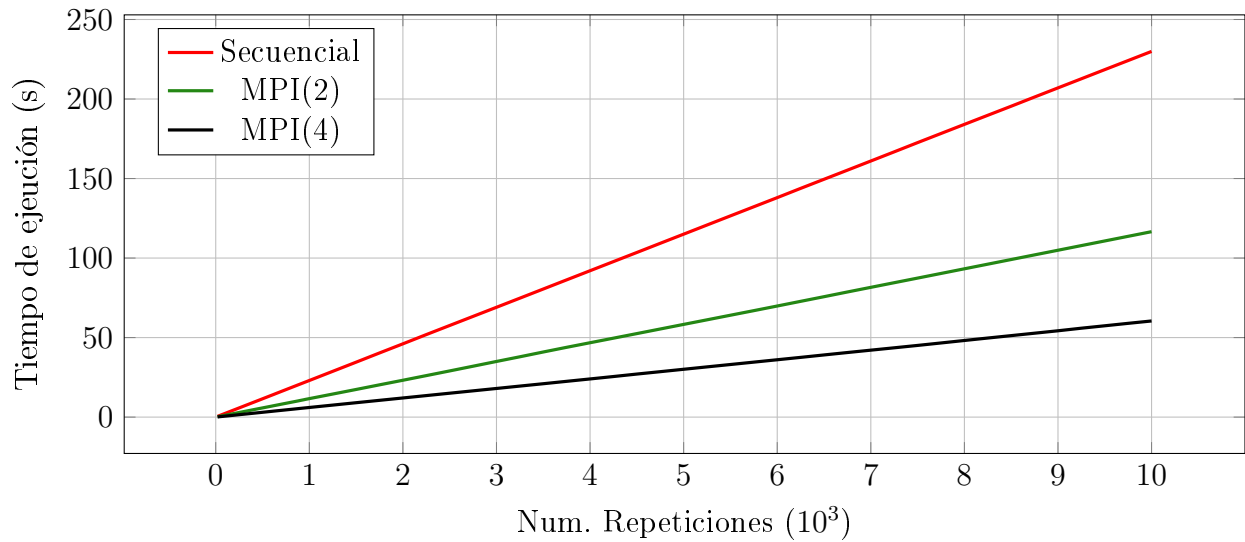


Figura 4.31: Tiempo de ejecución de la estrategia de dividir el trabajo de la Red Neuronal en ordenador de propósito general

La repartición de individuos es crucial para un correcto aprendizaje de la red.

No obstante, esta estrategia no converge en buenas predicciones. Hacer la media de los pesos de las neuronas obtenidos en cada proceso, no da buenos resultados. Si comprobamos la efectividad de una red sin entrenar, únicamente inicializados los pesos de manera aleato-

ria, se obtienen los mismos resultados que aplicando esta estrategia, es decir, predicciones erróneas. Este modelo de inteligencia artificial aprende en un proceso iterativo, y no se puede paralelizar con programación distribuida. Al menos en nuestras pruebas, no hemos logrado encontrar una combinación de hiper-parámetros que obtengan buenas predicciones.

Aunque no se pueda paralelizar el entrenamiento, debido a no poder obtener buenas predicciones, se puede aplicar este modelo de trabajar de manera paralela entre varios procesos para realizar una búsqueda exhaustiva de los mejores hiper-parámetros de la red neuronal. Para el algoritmo DQN de aprendizaje por refuerzo comentado en la Sección 3.3.2 se puede buscar los mejores hiper-parámetros para el entrenamiento del agente, y lograr buenos resultados en el entorno (*PacMan*).

Primero, hay que comentar los hiperparámetros que se van a estudiar. El algoritmo DQN tiene tres variables principales: el factor de descuento (γ), la variable de exploración-explotación (ϵ) y la tasa de aprendizaje. El factor de descuento es la variable que más estudios se han realizado llegando a la conclusión que -generalmente- el mejor valor es de 0.99 , como demuestra un estudio realizado en la universidad de Toronto¹⁶. Es por esto que la búsqueda solo se centra en los siguientes dos parámetros:

- *Epsilon decay*. Esta variable marca cuanto se reduce entre episodios la variable epsilon. Epsilon empieza con un valor de 1 , pues necesitamos que el agente aprenda correctamente a recoger las monedas sin chocar con los fantasmas. Se usa este valor para realizar una pequeña fase en la que el agente se mueve forma completamente aleatoria, y una vez ha aprendido a salir de los primeros estados, conviene ir reduciendo progresivamente el valor epsilon para seguir explorando el entorno.
- Tasa de aprendizaje (*learning rate*). Esta variable es fundamental para que el agente aprenda correctamente. Para ello se necesitan valores pequeños para que aprenda poco a poco a como completar el entorno en el que se encuentra.

Este algoritmo tiene dos redes neuronales, una utilizada para saber cuál es la mejor

acción para el estado actual, y otra para comprobar el estado siguiente, conocida como red neuronal objetivo. La segunda red se actualiza en menor frecuencia, pero ambas tienen los mismos parámetros. Una zona oculta con dos capas de 64 nodos, un valor de 64 para el número de ejemplos de entrenamiento que se utilizan para actualizar los parámetros de la red neuronal durante una sola iteración del entrenamiento, conocido como *batch size*. Y se usa el factor de descuento que se comentó anteriormente (0.99).

La prueba se ejecuta con nueve procesos, sin usar un proceso *master* el programa asigna a todos los procesos el mismo conjunto de ocho valores del hiper-parámetro *epsilon decay*, mientras que asigna un valor distinto de la tasa de aprendizaje para cada proceso. La Figura 4.32 muestra estos valores, siendo el primer conjunto las variables que van a ejecutar todos los procesos y el segundo conjunto las variables que se dividen entre los procesos de forma única. Cada proceso ejecuta en paralelo el algoritmo secuencial durante mil episodios. Al finalizar una ejecución del algoritmo, el proceso almacena en un fichero de texto la puntuación media de los últimos cien episodios, y si no ha terminado de procesar los datos, continúa a la siguiente ejecución.

```
eps_dec_vals = [1e-4, 2.5e-4, 5e-4, 7.5e-4, 1e-5, 2.5e-5, 5e-5, 7.5e-5]
lr_vals = [1e-3, 2.5e-3, 5e-3, 1e-4, 2.5e-4, 5e-4, 1e-5, 2.5e-5, 5e-5]
```

Figura 4.32: Valores de la búsqueda exhaustiva de los hiper-parámetros

La Figura 4.33 muestra en forma de malla 3×3 las ejecuciones en paralelo de los nueve procesos ejecutados. Cada histograma representa las ejecuciones de cada proceso, nombrando cada gráfica con su tasa de aprendizaje correspondiente. El *eje X* de cada histograma representa los valores de *epsilon decay*, comunes para todos los procesos, mientras que el *eje Y* la puntuación media de los últimos cien episodios ejecutados en un intervalo de $[-104.75, 313.95]$, siendo estos la puntuación mínima y máxima obtenida, respectivamente. Como se puede apreciar, la tasa de aprendizaje que mejores resultados obtiene es la primera, siendo este valor de $1e-3$, en el cual se obtiene la mejor puntuación además de ser el proceso

que más puntuaciones positivas obtiene. Observando las demás gráficas se puede ver una tendencia en los resultados, al aumentar la tasa de aprendizaje las puntuaciones empeoran. Aunque, en la sexta gráfica, con un valor de $5e-4$ se obtienen buenas puntuaciones. Esta búsqueda realiza en total 72 ejecuciones del algoritmo, pero al paralelizarse entre nueve procesos el tiempo de ejecución se reduce al proceso que más tiempo tarda en ejecutar las ocho ejecuciones, siendo este el primer proceso, pues el agente sobrevive más tiempo.

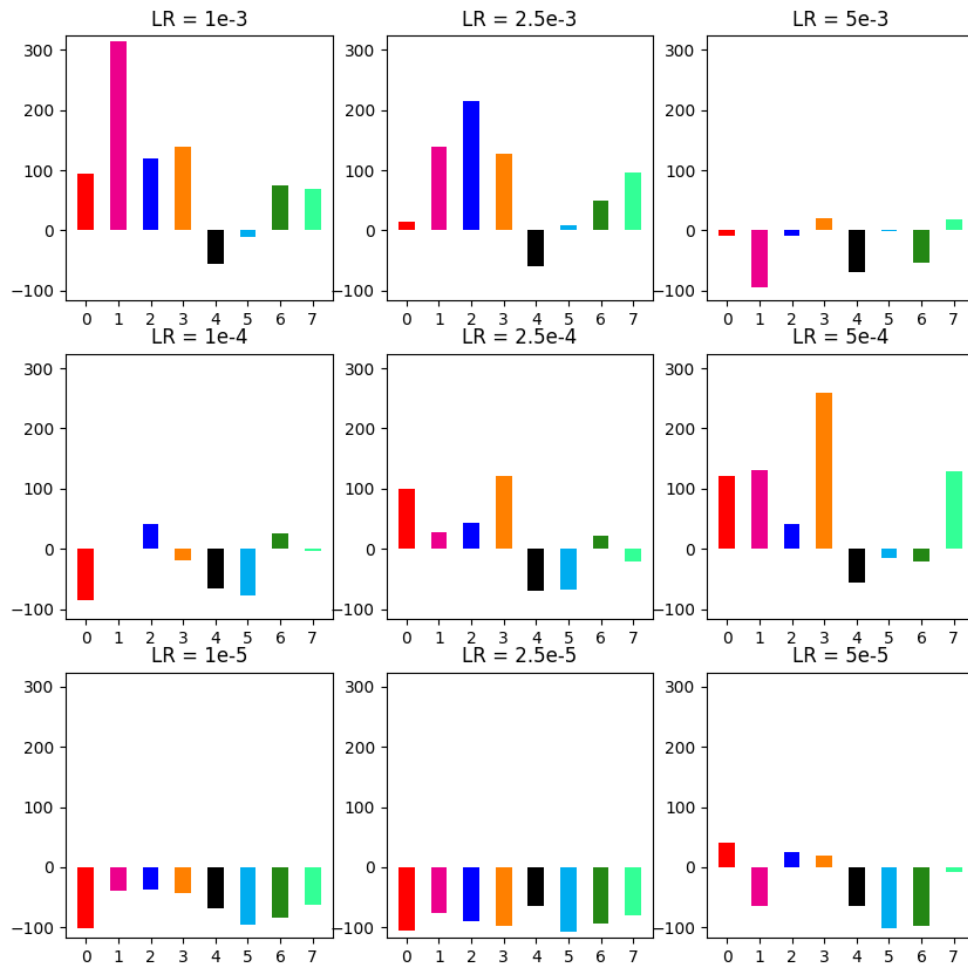


Figura 4.33: Búsqueda exhaustiva de parámetros del algoritmo DQN ejecutando nueve procesos

Capítulo 5

Conclusiones y trabajo futuro

En este trabajo se han desarrollado varias mejoras en distintos algoritmos de IA, a través de la biblioteca estándar de paso de mensajes MPI. Los desafíos encontrados durante su desarrollo resultaron ser más complejos de lo que se había previsto inicialmente. Los problemas de configuración de la biblioteca MPI en Windows, así como y adaptar la gestión de bibliotecas de Python usando Anaconda, fueron unos problemas completamente imprevistos. El desconocimiento general de MPI se debe a la ausencia de asignaturas específicas de programación distribuidas en el grado de Ingeniería Informática, únicamente ofreciendo fundamentos teóricos, sin profundizar en la práctica. La ejecución de las pruebas en el sistema distribuido de la Facultad de Informática, junto con la documentación del trabajo desarrollado, tomó más tiempo del calculado. Sin embargo, fue una etapa sumamente gratificante e interesante.

Una vez finalizadas las estrategias propuestas, se ha llevado a cabo una fase de experimentación, que ha consistido en analizar los tiempos de ejecución, variando tanto los parámetros disponibles, como los conjuntos de poblaciones para cada tipo de algoritmo. Cabe destacar el alto coste computacional de los experimentos. Por ejemplo, en el algoritmo jerárquico aglomerativo con distancia por enlace simple, una prueba en el sistema distribuido requirió más de un día en finalizar, obligando a reducir el tamaño de la población usada. Es importante remarcar que algunos de los resultados obtenidos no coincidían con la tendencia esperada. En particular, la estrategia de segmentación (pipeline) realizada en las redes

neuronales, donde el rendimiento alcanzado fue peor que el algoritmo secuencial a pesar de contar con -al menos- tres procesos. Aparte de disponer con los resultados obtenidos en la misma estrategia, pero para los algoritmos evolutivos, en los cuales si se obtuvieron buenos resultados. Después de realizar un análisis, observamos que la causa de estos resultados se debe a contar con dos flujos de mensajes en direcciones opuestas. La importancia de los buenos resultados es equivalente a obtener resultados no tan eficaces como se esperaban, pues es un avance para extraer conclusiones u otras ideas a implementar.

Una de las cosas más importantes que he aprendido a lo largo del trabajo es que no siempre *más es mejor*. Utilizar más procesos no tiene por qué derivar en un rendimiento proporcional al trabajo ejecutado. La sobrecarga (*overhead*) de los procesos en las implementaciones es un fundamento a tener en cuenta a la hora de ejecutar programas, y en la vida misma.

Como trabajo a futuro se propone investigar otros algoritmos de las técnicas desarrolladas, además de investigar y mejorar otras técnicas de IA, como puede ser el procesamiento del lenguaje natural.

Conclusions and future work

In this work, several improvements have been developed in different AI algorithms, through the standard MPI message passing library. The challenges encountered during its development turned out to be more complex than initially anticipated. The problems with configuring the MPI library in Windows, as well as adapting the management of Python libraries using Anaconda, were completely unforeseen problems. The general lack of knowledge of MPI is due to the absence of specific programming subjects distributed in the Computer Science degree, only offering theoretical foundations, without delving into practice. The execution of the tests in the distributed system of the Faculty of Informatics, together with the documentation of the work developed, took more time than estimated. However, it was an extremely rewarding and interesting stage.

Once the proposed strategies were finalized, an experimentation phase was carried out, which consisted of analyzing the execution times, varying both the available parameters and the sets of populations for each type of algorithm. It is worth highlighting the high computational cost of the experiments. For example, in the hierarchical agglomerative algorithm with single link distance, a test on the distributed system required more than a day to complete, forcing the size of the population used to be reduced. It is important to note that some of the results obtained did not coincide with the expected trend. In particular, the segmentation strategy (pipeline) carried out in neural networks, where the performance achieved was worse than the sequential algorithm despite having -at least- three processes. Apart from having the results obtained in the same strategy but for the evolutionary algorithms, in which good results were obtained. After performing an analysis, we observed that the cause of these results is due to having two message flows in opposite directions. The importance of good results is equivalent to obtaining results that are not as effective as expected, since it is an advance to draw conclusions or other ideas to implement.

One of the most important things I have learned throughout this work is that *more is not always better*. Increasing the computational resources does not always have a proportional impact on the overall system performance. The *overhead* of processes in implementations is a fundamental thing to take into account when executing programs, and in life itself.

As future work, it is proposed to investigate other algorithms of the developed techniques, in addition to investigating and improving other AI techniques, such as natural language processing.

Bibliografía

- [1] Bloom AI. Último acceso: 2024-02-08. <https://bloomai.co/>.
- [2] ChlouisPy - Maze generator. Último acceso: 2024-02-21. <https://github.com/ChlouisPy/maze-generator-maze-solver>.
- [3] Marcel R. Ackermann, Johannes Blömer, Daniel Kuntze, and Christian Sohler. Analysis of agglomerative clustering. *Algorithmica*, 69:184–215, 2014.
- [4] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: from merge sort to timsort. 2015.
- [5] Brandon Barker. Message passing interface (mpi). In *Workshop: high performance computing on stampede*, volume 262. Cornell University Publisher Houston, TX, USA, 2015.
- [6] Marco A. Contreras-Cruz, Víctor Ayala-Ramírez, and Uriel H. Hernandez-Belmonte. Mobile robot path planning using artificial bee colony and evolutionary programming. *Applied Soft Computing*, 30:319–328, 2015.
- [7] Flor A. Espinoza, Janet M. Oliver, Bridget S. Wilson, and Stanly L. Steinberg. Using hierarchical clustering and dendrograms to quantify the clustering of membrane proteins. *Bulletin of mathematical biology*, 74:190–211, 2012.
- [8] Frédérick Garcia and Emmanuel Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.
- [9] Henrik Jeppesen. Carbon tracker initiative. In *World Scientific Encyclopedia of Climate Change: Case Studies of Climate Risk, Action, and Opportunity Volume 1*, pages 63–69. World Scientific, 2021.

- [10] Keith Kirkpatrick. The carbon footprint of artificial intelligence. *Communications of the ACM*, 66(8):17–19, 2023.
- [11] Frances Y Kuo and Ian H Sloan. Lifting the curse of dimensionality. *Notices of the AMS*, 52(11):1320–1328, 2005.
- [12] Giuseppe Lugano. Virtual assistants and self-driving cars. In *2017 15th International Conference on ITS Telecommunications (ITST)*, pages 1–5. IEEE, 2017.
- [13] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. *Advances in Neural Information Processing Systems*, 36:53038–53075, 2023.
- [14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [15] Surender Mor, Sonu Madan, and Kumar Dharmendra Prasad. Artificial intelligence and carbon footprints: Roadmap for indian agriculture. *Strategic Change*, 30(3):269–280, 2021.
- [16] Silviu Pitis. Rethinking the discount factor in reinforcement learning: A decision theoretic approach. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 7949–7956, 2019.
- [17] Jeff Pool. Accelerating sparsity in the nvidia ampere architecture. *GTC 2020*, 2020.
- [18] José Jaime Ruz Ortiz. Multiprocesadores de memoria compartida y distribuida. Universidad Complutense de Madrid (UCM), 12/01/2016.
- [19] Mohd Shamrie Sainin. Best programming languages for AI. 2021.
- [20] Harold S Stone. *High-performance computer architecture*. Addison-Wesley Longman Publishing Co., Inc., 1990.

- [21] Christopher A Thomas and Xander Wu. How global tech executives view us-china tech competition. 2021.
- [22] Yi Wang, Kok Sung Won, David Hsu, and Wee Sun Lee. Monte carlo bayesian reinforcement learning. *arXiv preprint arXiv:1206.6449*, 2012.

Acrónimos

MPAI Message Passing Artificial Intelligence

AI Artificial Intelligence

MPI Message Passing Interface

CPU Central Processing Unit

GB Giga-Byte

RAM Random Access Memory

CO₂ Carbon Dioxide

HPC High Performance Computing

SPMD Single Program Multiple Data

RL Reinforcement Learning

MDP Markov Decision Process

DQN Deep Q-Network

KNN K-Nearest Neighbors

PEV Programación EVolutiva