Universidad de Concepción.
Dirección de Pregrado
Facultad de Ingeniería – Ingeniería Civil en Informática

Búsqueda de caminos en entornos multiagente, mediante ajustes restrictivos sobre búsquedas heurísticas.

Memoria para optar al grado de
INGENIERO CIVIL INFORMÁTICO

POR
Manuel Alejandro Díaz Ulloa
CONCEPCIÓN, CHILE
Agosto, 2020

Profesor Guía: Julio Godoy D.
Comisión: Lilian Salinas A.
José Fuentes S.

Agradecimientos.

A mi Pintas, mi familia, Leonardo y Patricio. Gracias

Bele

Resumen.

En esta memoria se presenta un estudio respecto al desempeño entregado por dos algoritmos de búsqueda de caminos, para entornos con múltiples agentes sobre escenarios discretizados. La principal motivación es lograr que cada agente llegue a su celda objetivo, evitando estancarse frente a otros agentes y minimizando el costo de su ruta a seguir.

El primer algoritmo (*Baseline*) es una implementación lo más próxima al algoritmo *BMAA** y el segundo algoritmo *Conflict Resolve - Multi Agent Pathfinding* (*CR-MAPF*) añade una capa predictiva, con la cual se determinan restricciones a respetarse sobre el camino de cada agente. Dado que al escalar la cantidad de agentes sobre *CR-MAPF* estos se estancan, se considera y evalúa el desempeño de añadir una modificación (*push_out*) sobre *CR-MAPF*, con el objetivo de lograr aumentar la cantidad de agentes que logren llegar a sus celdas de destino.

La recolección de datos realizada lleva a determinar que *CR-MAPF*, ve afectado su desempeño al incrementar la densidad de agentes sobre el mapa, esto principalmente debido a las restricciones y ajustes que realiza el algoritmo sobre cada agente, pero en cambio, *Baseline* en mapas adversos con mayor densidad provoca perdida de ciertos agentes, los cuales acaban por no llegar a sus celdas de destino.

Índice general.

Capítulo 1.

INTRODUCCIÓN.

Desde los inicios del desarrollo de la informática se ha buscado diseñar algoritmos para encontrar respuestas a problemas de uso común (búsqueda de elementos, ordenamiento, automatizar procesos, etc.), entre ellos uno de los problemas que más destaca es la búsqueda del camino mínimo entre un punto origen a uno de destino, el cual se encuentra ligado fuertemente al estudio de la teoría de grafos. Existen distintas maneras de abordar este problema, entre ellas el algoritmo de *Dijkstra*, su variante mediante búsqueda informada, el algoritmo *A**, entre otras. Todos estos algoritmos se enfocan en resolver el problema en el que solo un agente se desenvuelve en el ambiente y su implementación es poco eficiente o inviable al momento de escalar la cantidad de agentes que se desplazan sobre el mismo ambiente a la vez. Para ello existen distintas aproximaciones que abordan esta problemática y estas reciben el nombre de *Multi Agent Pathfinding Algorithms (MAPF)*. Sobre este conjunto de algoritmos existen aquellos que funcionan en tiempo de ejecución, otros que realizan un análisis previo del entorno y los que utilizan ambas técnicas para lograr que todos los agentes logren su cometido.

Los algoritmos *MAPF* se encuentran diseñados para encontrar el camino con el cual cada agente sobre el mapa llegue a su destino evitando colisionar con otros, en el que a su vez se optimice la función de costo de desplazamiento y se minimice la suma de las longitudes de los caminos de todos los agentes. Gran parte de estos algoritmos son una adecuación de *A** y su uso se extiende desde el campo de los videojuegos, donde en un escenario ficticio múltiples agentes se desplazan a la vez, hasta el área industrial, donde es posible mediante múltiples robots llevar a cabo la gestión del almacenamiento en bodegas evitando colisiones entre estos.

1.1. Descripción del problema.

En la actualidad existen distintas adecuaciones del algoritmo *A** para la búsqueda de caminos sobre entornos con múltiples agentes, estos algoritmos presentan un desempeño similar a pequeña escala y en situaciones favorables (como un mapa abierto o bajo número de agentes en movimiento), pero al momento de escalar la cantidad de agentes y plantear un mapa adverso (con estrechos pasillos por ejemplo), estos presentan conflictos particulares por algoritmo, conflictos sobre los cuales si no se cuenta con la debida identificación y gestión de estos, pueden conllevar situaciones de interbloqueo o extender de manera innecesaria el camino asociado a cada agente.

Una representación real del escenario planteado es posible apreciarla con los robots *Kiva* utilizados por la compañía *Amazon* dentro de sus almacenes para el transporte de carga. Estos robots logran coordinar su movimiento en las dependencias del almacén en que se encuentran y su entorno se les es entregado de forma discretizada como una grilla. Esto permite que entre ellos les sea posible analizar la acción de los demás robots cercanos y en base al comportamiento de su entorno determinar qué acción realizar, pese a que el algoritmo utilizado por *Amazon* con este modelo en específico no es público, se conoce que los robots utilizan un sistema de puntajes con respecto a las estanterías con mayor afluencia o demanda [1], por lo cual existe coordinación entre los robots gestionada por un controlador principal, que provee a los agentes de las decisiones a tomar en cuenta.

Otro escenario de estudio que a su vez permite evidenciar el funcionamiento de los algoritmos de búsqueda de caminos, es el mundo de los videojuegos. En ellos se presentan entornos adversos, sobre los cuales el movimiento de los agentes se modifica en base al escenario y al movimiento de los demás agentes. Sobre la mayoría de estos escenarios (Por ejemplo, *Age of Empires*, *Total Annihilation*, entre otros), se utiliza una adecuación del algoritmo *A**, como lo es *RTAA**. El algoritmo *RTAA** utiliza las entradas de la ubicación, destino y visión del agente a controlar, para entregar una ruta que lo desplace en dirección a su destino.

En situaciones donde la cantidad de agentes es superior, debido al alto costo que significa encontrar el menor camino libre de colisiones al objetivo (considerando que otros elementos del escenario también se desplazan), el algoritmo *RTAA** en específico asume que las celdas fuera del rango de visión del agente no se encuentran bloqueadas,

permitiendo al agente desplazarse localmente sobre su rango de visión mediante *A** en dirección a su destino, lo cual evita calcular la ruta completa del agente desde un principio [2]. Sin embargo, *RTAA** exhibe problemas al aumentar la cantidad de agentes o al modificar parámetros como la visión que posee cada agente, estos problemas conllevan por lo regular a situaciones de interbloqueo[1] entre los agentes.

1.2. Objetivo principal.

El pie de página debería ir en el primer párrafo de

El objetivo principal de este trabajo es mejorar el desempeño de una implementación que hace uso de *RTAA**. Donde se busca lograr que por cada agente sea posible encontrar un camino libre de colisiones que lleven este a su celda de destino, esto mediante búsquedas heurísticas en tiempo de ejecución, sin necesidad de coordinación explicita entre los agentes.

1.2.1. Objetivos específicos.

- Explicar parte de los distintos algoritmos de búsqueda para un agente y *MAPF*.
- Documentar situaciones y comportamientos excepcionales entre los algoritmos, así como también dar explicación a estos sucesos.
- Escalar la implementación provista a un número de agentes significativo.
- Proveer una rutina para la gestión de interbloqueo entre agentes.
- Realizar una evaluación experimental en distintos escenarios.
- Lograr una evaluación comparativa de *CR-MAPF* con *Baseline*.

1.3. Alcances y limitaciones.

Ambas implementaciones aquí presentadas corresponden a algoritmos estocásticos. Las evaluaciones experimentales realizadas consideran una cantidad 40 ejecuciones completas por muestra, esto con tal de evitar sesgar los resultados a una única ejecución que resultase favorable para alguno de los algoritmos a ser comparado.

El presente estudio no busca realizar una medición de desempeño a nivel de hardware y tiempos de ejecución, pese a ello, la selección de inputs (mapas, variables, número de agentes) si se ve influenciada por el rendimiento de los algoritmos y por el hardware que se tenía a disposición.

Los resultados obtenidos se encuentran respaldados en su totalidad sobre el repositorio correspondiente a este trabajo.

1. Interbloqueo: situación sobre la cual dos agentes buscan utilizar un mismo recurso en simultaneo, el cual puede ser un estado o casilla.

Capítulo 2.

MARCO TEÓRICO.

Sobre esta sección se presentan las nociones básicas y algoritmos empleados, sobre estos se detalla su funcionamiento y se realiza un análisis comparativo respecto a los objetivos del presente estudio. Adicionalmente se detallan aspectos de la arquitectura y el entorno empleado para la realización de este trabajo.

2.1 Teoría de Grafos.

La teoría de grafos tiene origen de las matemáticas discretas y matemáticas aplicadas. Su influencia abarca distintos y variados campos de estudio como lo son las ciencias sociales, psicología, telecomunicaciones, ciencias de la computación, etc.

Los grafos son estructuras matemáticas compuestas por vértices y aristas, utilizadas para representar ~~parejas de~~ relaciones entre objetos o conceptos [11], por ejemplo, un grafo $G = (V, E)$ se compone por el par $V = \{v_1, \dots, v_n\}$ (V como conjunto no vacío de vértices) y $E = \{e_1, \dots, e_m\}$ (E como conjunto de aristas). Sobre cada elemento del conjunto E, se identifica un e el cual corresponde a una arista que conecta dos vértices sobre el conjunto V. Adicionalmente un grafo ponderado es aquel donde el conjunto adicional $W = \{w_1, \dots, w_m\}$ (de igual tamaño que E) identifica el peso asociado a la arista, tal que $\forall e_i \in E, \exists! \, w_i \in W$, con $i \in \{1, \dots, m\}$. Asimismo W es posible representarse mediante una función de asignación $w: E \to \mathbb{R}$, lo cual significa que $\forall e \in E$, su peso esta dado por $w(e) \in \mathbb{R}$.

En línea con el cu

2.2 Problema del camino mínimo.

Dentro de la teoría de grafos el problema del camino más corto o problema del camino mínimo consiste en encontrar un camino entre dos vértices, tal que la suma de las aristas que constituyen ~~el~~ este camino sea la menor posible. Sobre un grafo $G = (V, E)$, un camino es una secuencia $P = (v_1, \dots, v_n) \in Vx \dots x\, V$, tal que el vértice v_i es adyacente a v_{i+1} para $i \in \{1, \dots, n\}$.

Para ser un poco más precisos, cambiaría la definición de P. Actualmente, P y V están definidos sobre vértices con sub

Se nombra $e_{ij} \in E$, a la arista incidente en ambos vértices v_i y v_j dentro de un grafo no dirigido. Entonces dada una función de costo $w: E \to \mathbb{R}$, un grafo simple $G = (V, E)$, el camino mínimo de v a v' es aquella secuencia $P = (v_1, v_2, \dots, v_n)$ (donde $v_1 = v$ y $v_n = v'$), tal que el n seleccionado minimiza la suma $\sum_{i=1}^{n-1} w(e_{i,i+1})$. En tal caso que cada arista tenga un peso unitario nuestra función de costo siempre mapeara lo valores de la arista de la siguiente manera $w: E \to \{1\}$.

Este es el planteamiento sobre el problema del camino mínimo a tratar en el presente documento.

2.3 Algoritmos para la búsqueda de caminos para un único agente.

Los algoritmos de búsqueda de caminos usualmente se emplean para resolver el problema del camino mínimo planteado sobre la teoría de grafos. Este tipo algoritmos se emplean en distintas aplicaciones como lo son los sistemas de navegación satelital, enrutar paquetes a través de internet, aplicaciones de mapeado o en videojuegos.

Los siguientes algoritmos encuentran solución al problema del camino mínimo en tiempo polinomial para un único agente.

2.3.1 Algoritmo de Dijkstra.

El algoritmo de *Dijkstra* permite determinar el camino mínimo desde un vértice $v \in V$ hacia el resto de los vértices sobre un grafo ponderado y conexo. Este algoritmo consiste en explorar todos los caminos mínimos que comienzan del vértice v y que llevan a los demás vértices.

Formalmente, sobre el grafo ponderado conexo $G = (V, E)$, con s como nodo inicial y D el vector de distancias de tamaño $|V|$, donde $D[i]$ es el coste del camino mínimo conocido desde s hasta i. El algoritmo *Dijkstra* que hace uso de una cola de prioridad como estructura de datos adicional es el siguiente [3].

Algorithm 1: Dijkstra$(G(V,E),s)$

```
1  Array D[|V|]
2  Array parent[|V|]
3  Array visited[|V|]
4  minHeap Q
5  for i ← 0 to |V| do
6  |   D[i] ← Infinito
7  |   parent[i] ← −1
8  |   visited[i] ← false
9  D[s] ← 0
10 Q.push((s, D[s]))
11 while Q contenga elementos do
12 |   u ← Q.pop()
13 |   visited[u] ← true
14 |   for v ∈ G.adj[u] do
15 |   |   if not visited[v] then
16 |   |   |   if D[v] > D[u] + peso(u, v) then
17 |   |   |   |   D[v] ← D[u] + peso(u, v)
18 |   |   |   |   parent[v] ← u
19 |   |   |   |   Q.push((v, D[v]))
20 return D
```

|V|-1

Lo más correcto es Q.extract_min()

Esa condición no va en Dijkstra

Ojo acá. Q.push() inserta un nuevo par, pero lo que realmente necesit

El primer ciclo iterativo inicializa los arreglos D (distancia de s a i), $parent$ (nodos que preceden al elemento i) y $visited$ (nodos que ya se han visitado).

elemento con me

Luego la distancia de s a si mismo se marca como cero y se inserta sobre el $minHeap$[2], entonces dentro del ciclo $while$ se extrae el primer elemento del $minHeap$, se marca como visitado y se itera sobre los nodos adyacentes a este. Si el nodo adyacente no se encuentra visitado y la distancia que se encuentra en memoria es mayor que la del nodo extraído más el peso hasta el adyacente, entonces se ajusta el valor en memoria, se escribe el nodo que precede dentro de $parent$ y se inserta este dentro del $minHeap$.

Esa condición no

Al finalizar, sobre cada casilla $D[i]$ está contenida la distancia mínima desde s a i.

El algoritmo de $Dijkstra$ presenta las bases sobre las que se diseñó el algoritmo *A**, el cual es esencial para el presente documento.

2.3.2 Algoritmo de búsqueda A*.

El algoritmo *A** permite encontrar el camino más corto sobre grafo informado, este mediante una función de evaluación heurística permite expandir aquellos nodos que presenten una menor distancia desde ese nodo al nodo de destino, a diferencia de

2. MinHeap: Es un árbol binario completo, que cumple con la propiedad de conservar el menor valor ingresado cómo la raíz.

No es del todo precisa esa definición, ya que para cada sub-árbol también se cumple que

cómo sería resuelto por un algoritmo *greedy*[3] que busque el nodo vecino con el menor costo inmediato. El procedimiento consiste en utilizar la función de evaluación $f(n) = g(n) + h'(n)$, donde $h'(n)$ representa el valor desde el nodo actual n hasta el nodo de destino y g(n) el costo del camino conocido desde el nodo de origen al nodo n.

Explicar ¿en términos de qué se mide e

Considerando el grafo $G(V,E)$, el nodo inicial s, el nodo de destino $goal$, el vector de nodos expandidos $Open$ y cerrados $Close$, los pasos a seguir son los siguientes [4].

Algorithm 2: A-star($G(V,E)$,s,$goal$)

```
1  Se instancia openList y closedList, ambas como una lista vacía de nodos
2  Se inserta s sobre openList
3  while openList contenga elementos do
4      El nodo v se toma como el que contenga menor f sobre openList.
5      Se extrae v de openList.
6      Se añade v a closedList.
7      if v es goal then
8          Se encontró la solución.
9          reconstruimos el camino mediante vuelta atrás.
10         Se acaba la ejecución.
11     generamos los nodos adyacentes v.
12     for cada nodo u adyacente a v do
13         if u se encuentra en closedList then
14             Evitamos este nodo y pasamos al siguiente nodo u.
15         u.g ← v.g + distancia entre u y v
16         u.h ← distancia de u a goal
17         u.f ← u.g + u.h
18         for n ∈ openList do
19             if u == n y u.g > n.g then
20                 Evitamos este nodo y pasamos al siguiente nodo u.
21         Se añade u a openList.
```

2.3.3 Esquema comparativo.

Adicionalmente a ambas aproximaciones presentadas se considera la búsqueda *Best-First-Search* para esquematizar un ejemplo del desempeño de cada algoritmo. Esta búsqueda consiste en encontrar aquel nodo adyacente sobre el cual se minimice el costo de desplazamiento localmente, esto no asegura un camino mínimo dado que no se presta atención en que la ruta que sigue la expansión del siguiente nodo es la correcta para lograr este cometido.

3. Algoritmo greedy: Algoritmos que realizan búsquedas óptimas locales, para hallar un resultado.

La figura 2.1 evidencia como desde el nodo de origen (naranja) al nodo de destino (lila), el algoritmo *Best-First-Search*, es capaz de obtener un camino válido al nodo objetivo (línea blanca segmentada), pero no logra minimizar su coste de desplazamiento.

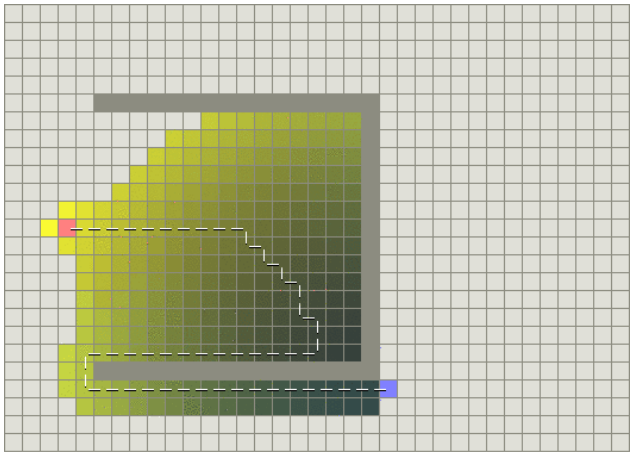


Figura 2.1: Ejemplo búsqueda Best-First-Search [9]

Por otra parte, la ejecución del algoritmo de *Dijkstra* (Figura 2.2) permite obtener un camino mínimo válido al nodo objetivo, pero la cantidad de nodos expandidos o visitados se incrementa cuantiosamente (nodos con color degradado de negro a azul).

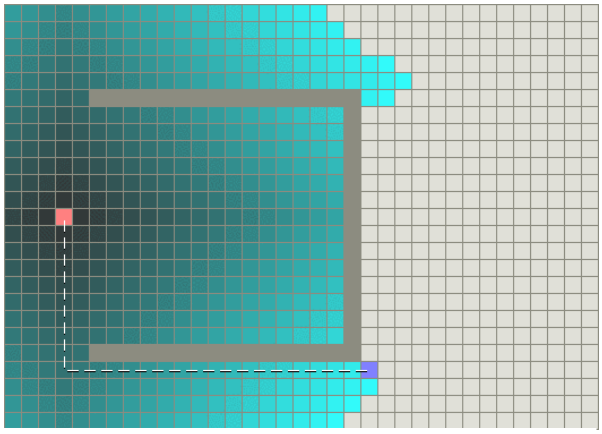


Figura 2.2: Ejemplo búsqueda Dijkstra [9]

Mientras que para el algoritmo de búsqueda informada *A**, le es posible obtener el camino mínimo evitando expandir una alta cantidad de nodos sobre el entorno como lo si lo hace el algoritmo de *Dijkstra*.

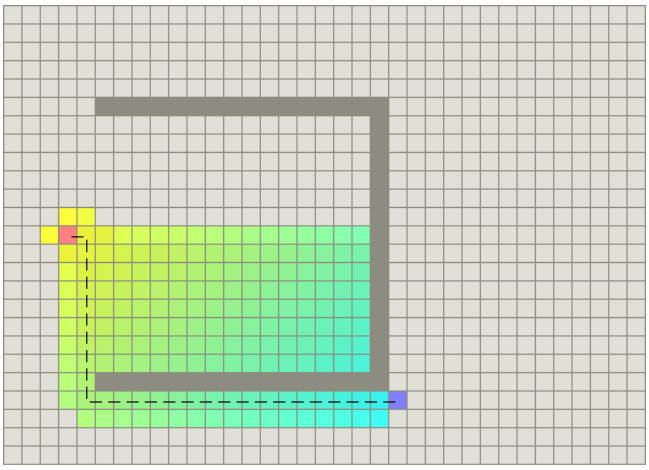


Figura 2.3: Ejemplo búsqueda A [9]*

Comparativamente las complejidades de estos algoritmos en peor caso son para *Best-First-Search* (con enfoque *DFS* o *BSF*) de $\boldsymbol{O}(|\boldsymbol{E}| + |\boldsymbol{V}|)$, lo cual corresponde con el total de vértices y aristas expandidas producto de la búsqueda inmediata de la arista con menor costo. Luego para el algoritmo de Dijkstra mediante *MinHeap* es $\boldsymbol{O}(|\boldsymbol{E}|\boldsymbol{log}(|\boldsymbol{V}|))$, esto debido a los nodos adyacentes expandidos producto de las extracciones sobre el *MinHeap* y a la cantidad de aristas que se evalúan para ajustar la distancia desde el nodo de origen. Mientras que para *A** su complejidad es de $\boldsymbol{O}(|\boldsymbol{E}|)$, considerando que la heurística entregada es consistente.

Revisar la compl

2.4 Algoritmos para la búsqueda de caminos multiagente (MAPF).

En un problema de búsqueda de caminos multiagente, se busca encontrar los caminos libres de colisiones para un conjunto de agentes sobre un mapa, dado un punto de origen y de destino para cada agente. Entre la variedad de algoritmos propuestos para este problema, es posible distinguir entre algoritmos exactos y heurísticos. Sobre estos últimos se encuentran aquellos que trabajan en tiempo de ejecución, aquellos que trabajan calculando previamente las rutas a seguir y los que utilizan ambos enfoques para dar solución al problema.

El problema *MAPF* se compone por el par (G, A), donde $G = (N, E, w)$ es un grafo ponderado no dirigido, con N el conjunto de nodos, $E \subseteq N \times N$ el conjunto de aristas e,

$w: E \rightarrow \mathbb{R}$ la función de costo y $A = \{a^1, \dots, a^n\}$ el set de agentes presentes, donde cada agente $a^i \in A$ es identificado mediante el par $(n^i_{start}, n^i_{goal})$, donde n^i_{start} es el nodo de inicio y n^i_{goal} es el nodo objetivo.

Por convención es de asumir que, sobre un espacio discretizado en grilla, cada nodo tendrá 8 celdas vecinas, pero sobre esta investigación se consideraran únicamente los movimientos realizados las direcciones norte, sur, este y oeste.

2.4.1 Algoritmos de búsqueda heurística en tiempo real.

En escenarios donde se requiere que los agentes comiencen a desplazarse de manera inmediata, resulta costoso computar el camino en ventanas de tiempo definidas, ya sea por que las dimensiones del escenario no permiten evaluar cada una de las combinaciones sobre los posibles caminos o existen elementos sobre el escenario que no dependen de las decisiones que ejecute el algoritmo. Por ello, los siguientes algoritmos buscan llevar a cada agente a su destino procurando que el desplazamiento comience de manera inmediata.

2.4.2.1 Real Time Adaptative A (RTAA*)*

*RTAA** realiza múltiples búsquedas *A**, estas mediante un rango de visión $lookahead$[4] previamente definido, por ello este algoritmo logra que las futuras búsquedas se encuentren más informadas respecto a iteraciones previas.

El procedimiento que sigue *RTAA** es el siguiente: por cada estado s presente en cada búsqueda A* a realizarse, se conservan dos elementos, el primero $g[s]$ que enuncia el costo mismo desde el estado inicial al estado s y el valor estimado al nodo de destino $f[s] = g[s] + h[s]$ a través del nodo s. Se asume que s se encuentra dentro del rango de $lookahead$ y este es expandido durante la búsqueda *A**. Es posible obtener un valor admisible $gd[s]$, el cual no sobreestime la distancia al objetivo de la siguiente manera; La distancia desde el estado actual s_{curr} a cualquier objetivo a través del estado s, es igual a la distancia desde s_{curr} al estado s más la distancia al objetivo $gd[s]$ desde el estado s. Este valor claramente no puede ser menor al valor $gd[s_{curr}]$ hasta el nodo objetivo desde el estado s_{curr}, esto por desigualdad triangular. Por ende, es posible establecer la siguiente ecuación.

$$g[s] + gd[s] \geq gd[s_{curr}]$$
$$gd[s] \geq gd[s_{curr}] - g[s]$$
$$gd[s] \geq f[\bar{s}] - g[s]$$

4. $lookahead$: Corresponde al máximo de celdas a expandirse producto de realizar una búsqueda *RTAA**

Donde $f[\bar{s}]$ corresponde al camino ideal desde el estado actual al objetivo. Consecuentemente el valor $f[\bar{s}] - g[s]$ provee una estimación admisible de la distancia al estado objetivo $gd[s]$ desde el estado s y esta puede determinarse rápidamente, por ello para lograr heurísticas mayormente informadas se calcula y asigna esta diferencia a cada estado por expandir durante la búsqueda *A** y por lo tanto esta información entra en la cola de estados sin expandir al acabar la búsqueda *A** [2].

En síntesis, *RTTA** realiza iterativamente búsquedas *A** sobre el rango dado por el $lookahead$, para luego continuar iterando hasta que el nodo de destino entre sobre el rango de $lookahead$ y el agente se alcance su objetivo con esta última búsqueda *A**.

El algoritmo que presenta el funcionamiento de *RTAA** es el siguiente.

Constantes y funciones:

- S: conjunto de estados sobre los que se realiza la búsqueda.
- $Goal$: conjunto de estados objetivos.
- $A()$: conjunto de acciones por cada estado.
- $succ()$: Función sucesor, obtiene el estado siguiente a partir del actual y la acción.

Variables:

- $lookahead$: Numero de estados máximos a expandirse. Valor positivo mayor a cero.
- $movements$: Número máximo de acciones a ejecutar. Valor positivo mayor a cero.
- s_{curr}: Estado actual del agente.
- c: costo actual de la acción a realizar.
- h: valor heurístico actual sobre el estado.
- g: Conjunto de valores *g*.
- $CLOSED$: Lista de estados sin expandir.
- $\overline{s}$: Estado a expandirse al acabar la búsqueda *A**.

Algorithm 3: RTAA-star()

```
1  while s_curr ∉ goal do
2      A-star(lookahead)
3      if s̄ == FAILURE then
4          Fallo al encontrar siguiente estado a expandir
5      for cada s ∈ CLOSED do
6          h[s] = g[s̄] + h[s̄] − g[s]
7          moves ← movements
8          while s_curr ≠ s̄ AND moves > 0 do
9              a ← acción dentro de A(s_curr) que minimiza el costo desde s_curr a s̄
10             s_curr ← succ(s_curr, a)
11             moves ← moves − 1
12             incrementa los deseos sobre s con c[s, a], donde s ∈ S y a ∈ A(s)
13             if cualquier incremento sobre c[s, a] esta sobre el trayecto s_curr a s̄ then
14                 Existe otro agente sobre el un estado entre s_curr y s̄
15                 break
```

Es común que al emplear *RTAA** sobre escenarios con múltiples agentes, se utilice un límite de $movements$ igual a uno, esto dado que permite al agente acceder a su siguiente celda sobre el trayecto, sin capitalizar el dominio de las celdas subsiguientes, permitiendo que sobre una iteración cada agente se desplace solo una vez. Bajo este escenario las últimas rutinas presentes sobre la línea 12 a la 15 corresponden a una verificación del estado sucesor desde s_{curr}, verificando si el desplazamiento a este es válido o no.

Es necesario precisar que *RTAA**, a diferencia de *A**, no asegura su completitud para encontrar siempre la solución deseada, por ejemplo, en escenarios en los que las vías de desplazamiento sean en un único sentido y estas lleven a un punto de no retorno para alcanzar el estado objetivo, o bloquearse mutuamente entre agentes evaluando cada uno de ellos la celda del otro agente como la celda esperada a desplazarse.

*2.4.2.2 BMAA**

*BMAA** es un algoritmo *MAPF* [8], en el cual cada agente ejecuta una copia modificada de *RTAA**. Este trabaja en tiempo de ejecución, pierde solo un poco de precisión al relocalizar los agentes y no requiere una coordinación explicita entre agentes, coordinar el control de todos los agentes o preprocesar el escenario a operar.

Sobre *BMAA** se identifican los siguientes parámetros relevantes.

- $expansions$ - Número límite de nodos a expandirse por búsqueda *A** sobre *RTAA**. Cumple la misma función que el parámetro *lookahead* empleado con *RTAA**.
- $vision$ - Distancia sobre la cual cada agente es capaz de observar otro agente.
- $moves$ - Número de movimientos que realiza cada agente sobre su camino antes que *RTAA** determine un nuevo camino para él.
- $push$ - valor booleano que índica si el agente es capaz de desplazar temporalmente a otro agente para alcanzar su celda de destino.
- $flow$ - valor booleano que índica si *RTAA** utiliza anotaciones de flujo.

Mientras que las etapas del algoritmo son las siguientes.

Procedimiento NPC-Controller.

En un comienzo se itera por cada agente realizando la llamada al procedimiento $Search - Phase$, esto para encontrar un prefijo del camino desde el nodo actual hacia el nodo de destino. En el paso de ejecución siguiente el controlador itera por todos los agentes consultando el nodo al cual el agente debiera moverse sobre el siguiente paso, que es el nodo siguiente sobre la ruta actual del agente, si el agente actual se encuentra bloqueado por otro agente que ya ha alcanzado su destino y el parámetro $push$ es verdadero, entonces el agente desplaza temporalmente al otro agente para alcanzar su objetivo, luego el agente que bloquea retorna a su nodo objetivo en las ejecuciones posteriores, esto debido a que todos los agentes se encuentran ejecutando *RTAA**. Finalmente, el agente llega a su nodo deseado e incrementa su actual contador de tiempo (figura 2.4).

Algorithm 2 BMAA*'s NPC Controller.

```
1:  procedure NPC-CONTROLLER(A)
2:      for all a^i ∈ A do
3:          a^i.Search-Phase()
4:      end for
5:      for all a^i ∈ A do
6:          if a^i.P(n^i_curr) is defined then
7:              n ← a^i.P(n^i_curr)
8:              if push ∧ n is blocked by agent a^j then
9:                  a^j.PushAgent()
10:             end if
11:             if n is not blocked by an agent then
12:                 a^i.MoveTo(n)
13:             end if
14:         end if
15:     end for
16:     time ← time + 1
```

Figura 2.4: Algoritmo NPC controller, destinado a secuenciar la búsqueda por cada agente. [8]

Procedimiento Search-Phase

Su ejecución permite al agente hallar un prefijo del camino desde su nodo actual hasta su nodo objetivo (figura 2.5). El procedimiento consiste en verificar si no existe un prefijo previo para el agente en su estado actual y si no se excede el límite de búsqueda, de darse esta situación se procede llamar al procedimiento de búsqueda *Search*. Luego si la búsqueda entrega un resultado no nulo, entonces se expande el primer estado objetivo *n* y se ajusta el valor heurístico *f* en función de *n*, para luego acabar actualizando los valores heurísticos de los estados cerrados (dado que si *expansions* es mayor a uno no se recalculará el prefijo sobre la iteración siguiente) y se finaliza incrementando el contador de movimientos.

Algorithm 3 BMAA*'s Search Phase.

```
1:  procedure SEARCH-PHASE
2:      if Search.P(n^i_curr) is undefined or time ≥ limit then
3:          Search()
4:          if Search.open ≠ ∅ then
5:              n ← Search.open.First()
6:              f ← g(n) + h(n)
7:              Update-Heuristic-Values(Search.closed, f)
8:              limit ← time + moves
9:          end if
10:     end if
```

Figura 2.5: Algoritmo Search Phase, destinado a encontrar un prefijo de camino sobre un agente. [8]

El ajuste de las heurísticas (figura 2.6) sobre la lista de nodos cerrados consiste en sustraer de *g(n)* a *f* y almacenar estás para que puedan ser accedas posteriormente.

Algorithm 4 BMAA*'s Update Phase.

```
1:  procedure UPDATE-HEURISTIC-VALUES(closed, f)
2:      for n ∈ closed do
3:          h(n) ← f − g(n)
4:      end for
```

Figura 2.6: Algoritmo Update Phase. [8]

Procedimiento Search

El procedimiento de búsqueda utiliza una adaptación de *A** (figura 2.7) con las siguientes modificaciones, primero cada agente conserva sus valores heurísticos a través de cada búsqueda realizada, segundo, la búsqueda acaba al abarcar la cantidad de nodos $expansions$ dentro de la lista de estados. Por ende, la ruta obtenida resultado de seguir a los nodos padres desde el nodo que se encuentra a punto de expandirse al nodo actual, es ahora solo el prefijo de una ruta desde el nodo actual del agente hasta su nodo objetivo.

Sobre el siguiente algoritmo es necesario identificar lo siguiente:

- P: Es el prefijo que contiene los estados producto de la búsqueda.
- exp: Servirá de contador para las expansiones realizadas.
- $open$: Contiene los estados que aún quedan por expandir

En tanto al procedimiento, este consiste en que mientras queden nodos por expandir, si se ha superado el límite de expansiones o la búsqueda alcanzo el estado objetivo, entonces se finaliza la ejecución, almacenando el prefijo del camino sobre P, en caso contrario, se busca el siguiente estado a expandirse y se itera sobre sus estados vecinos, si este estado vecino se encuentra bloqueado o fuera del rango de visión, se salta al siguiente estado vecino. Sino entonces se verifica que este no se encuentre en la lista de cerrados ni en la de pendientes por expandir, de ser así es porque este estado no se ha tomado en cuenta para visitarse, por ende, se evalúa

si es posible ajustar su valor heurístico $g(n)$ y se procede a añadir a los pendientes por visitar.

Algorithm 5 BMAA*'s Version of A*.

```
1:  procedure SEARCH
2:      P ← ()
3:      exp ← 0
4:      closed ← ∅
5:      open ← {n^i_curr}
6:      g(n^i_curr) ← 0
7:      while open ≠ ∅ do
8:          if open.First() = n^i_goal ∨ exp ≥ expansions then
9:              calculate P
10:             break
11:         end if
12:         n ← open.Pop()
13:         closed.Add(n)
14:         for n′ ∈ n.GetNeighbors(flow) do
15:             d ← distance(n^i_curr, n′)
16:             if n′ is blocked by an agent ∧ d ≤ vision then
17:                 if n′ ≠ n^i_goal then
18:                     continue
19:                 end if
20:             end if
21:             if n′ ∉ closed then
22:                 if n′ ∉ open then
23:                     g(n′) ← ∞
24:                 end if
25:                 if g(n′) > g(n) + c(n, n′) then
26:                     g(n′) ← g(n) + c(n, n′)
27:                     n′.parent ← n
28:                     if n′ ∉ open then
29:                         open.Add(n′)
30:                     end if
31:                 end if
32:             end if
33:         end for
34:         exp ← exp + 1
35:     end while
```

Figura 2.7: Algoritmo Search (búsqueda A). [8]*

Capítulo 3.

CUERPO PRINCIPAL.

En el presente capítulo se detallan el funcionamiento de las implementaciones provistas para la realización de este proyecto, se resumen las modificaciones realizadas respecto a los desafíos planteados y se exhiben escenarios particulares de comportamiento. Cabe mencionar que ambas implementaciones aquí mencionadas ejecutan un proceso estocástico, esto debido que, al realizar la evaluación del siguiente nodo a expandirse, se realiza sobre una dirección aleatoria en un comienzo, para luego continuar con las demás direcciones posibles. El utilizar el enfoque anterior mencionado conlleva a ciertas situaciones particulares (como dos caminos *A** igualmente factibles) que se detallaran sobre este capítulo.

3.1. Implementación Baseline.

La primera implementación para tratar corresponde a una aproximación del algoritmo *BMAA**. El código fuente provisto se encuentra escrito en *C* y posee ciertas diferencias con respecto a *BMAA**. Entre las diferencias se encuentran que no posee anotaciones de flujo y que la cantidad de movimientos por iteración se encuentra limitada a uno.

3.2 Implementación CR-MAPF.

La segunda implementación, utiliza la base propuesta por el algoritmo *Baseline* y añade una capa de verificación de conflictos mediante predicciones realizadas por sobre el camino de cada agente (Anexo 7), esto se logra manejando las observaciones que realiza cada agente sobre los demás agentes de su entorno y aplicando restricciones sobre el camino que maneja cada agente, mediante los roles que se designan de manera dinámica.

El procedimiento se puede entender mediante las siguientes etapas.

3.2.1 Inicialización.

Se inicializa el mapa, los costes heurísticos por cada agente, las ubicaciones de inicio y destino, entre otros elementos relevantes. Durante esta etapa cada agente realiza una visualización de su entorno dentro del rango $lookahead$, identificando cada agente que

se encuentre dentro de su rango de visualización junto con la celda que este mantiene bloqueada.

3.2.2 Ejecución.

Esta etapa se mantiene mientras existan agentes sin llegar a su celda de destino o se exceda el limite de ventanas de tiempo permitidas. En este instante por cada agente se ejecutan las siguientes rutinas.

3.2.2.1 Cálculo de camino A y determinación de conflictos.*

En primer lugar, se determina localmente el camino *A** con límite de $lookahead$, ignorando posibles colisiones. Adicionalmente durante esta etapa se determina el valor de $formula$ y con esta los costos sobre las celdas en conflicto. El valor de $formula$ se utiliza para priorizar el desplazamiento de los agentes que se encuentren a una mayor distancia de la última celda en la que les fue posible desplazarse en más de dos direcciones, esto ya que al atravesar por pasillos estrechos el agente asume un alto riesgo al desplazarse, por ello se almacena por cada agente esta celda y se determina el valor $formula$ para el agente i mediante la siguiente ecuación.

$$formula = h(position[i], i) + ((2 * distancia(lastMobileCell[i])) + 3)$$

Donde.

- i: Es el agente.
- $position[i]$: Es la celda actual sobre la que se posiciona el agente i.
- $h(position[i], i)$: Corresponde a la distancia heurística desde la celda $position[i]$ para el agente i, hasta la celda objetivo.
- $lastMobileCell[i]$: Corresponde a la última celda que permitió el movimiento en más de dos direcciones para el agente i.
- $distancia(lastMobileCell[i])$: Determina la distancia a la última celda que permitió el movimiento en dos direcciones para el agente i.

Luego de determinar el valor de $formula$ se determina por cada expansión realizada durante el proceso *A** los costos de conflicto sobre la nueva celda añadida al camino, estos costos se determinan mediante tres posibles situaciones de conflicto:

1. Que exista otro agente j que desee posicionarse sobre la celda expandida del agente i durante el mismo instante, es decir, un agente ingresa al camino del agente i y se prevé una colisión entre estos. Comúnmente este esta situación

Una duda: Las situaciones de conflicto 2 y 3 ¿son iguales si se considera un lookahead de 1?

se conoce como un conflicto sobre el vértice (**Vertex conflict**), dado que ambos caminos buscan utilizar la misma celda en el mismo instante.

2. Que exista previamente otro agente j sobre la celda expandida en el camino de i en una etapa futura, un instante previo a que i se posicione sobre ella. Este problema se conoce como conflicto de arista (**Edge conflict**), ya que el j se desplaza por el camino de i provocando conflicto en múltiples etapas.
3. Existe otro agente j que en el instante actual mantiene la celda expandida bloqueada.

De darse alguno de los casos anteriores se evalúa el agente que posea mayor $formula$, en caso de que j posea mayor $formula$ significa que la celda actual expandida es una celda de conflicto, ya que otro agente con mayor $formula$ desea acceder a esa celda. Una vez finalizada la evaluación de cada caso, si la celda expandida corresponde a una celda de conflicto entonces se determina el costo de conflicto para ella, el procedimiento es el siguiente.

$$conflictCost = \frac{1}{future - pathlength[i] + 2}$$

Donde.

- $pathlength[i]$: Es el tamaño actual del camino calculado incluyendo la celda actual expandida.
- $future$: Corresponde al instante futuro en el que se prevé la situación de conflicto, este parámetro se evalúa desde $pathlength[i]$ hasta $lookahead$.

El valor de $conflictCost$ es siempre positivo y se encuentra entre 0 y 1. De esta manera mientras menor sea la diferencia entre el instante en que se espera el conflicto ($future$) y el tamaño actual del camino ($pathlength[i]$), mayor será el costo de conflicto asociado a esa celda.

Una vez finalizada esta etapa se contará con el camino *A**, los costos de conflicto y la $formula$ para el agente i.

3.2.2.2 Cálculo de camino con restricciones.

Durante esta segunda etapa se aplican las restricciones sobre las celdas presentes en el camino del agente i. En esta etapa es necesario saber que si sobre una celda el valor

de $conflictCost$ este es cercano a 1, significa que el instante en el que se espera la situación de conflicto es más próximo que en el caso que este sea cercano a 0.

El procedimiento consiste en evaluar cada celda del camino y descartar aquellas que significan un riesgo muy alto mediante las siguientes condiciones.

- Si el $conflictCost > 0.49$, se descarta esa celda del camino, esta significa un riesgo de conflicto muy alto ya que la situación es muy próxima, porque la diferencia entre el largo del camino y la etapa en que se espera el conflicto es muy baja.
- Si $0.01 < conflictCost \leq 0.49$, el $celda.grado < 3$, entonces a menos que fuese el estado inicial el agente no debiera acceder a esta celda, dado que restringe su movilidad y se acaba descartando la celda del camino.
- Si $0.201 < conflictCost \leq 0.49$, el $celda.grado \geq 3$, esto significa que probablemente exista un agente sobre esa celda, pero el riesgo es menor y el agente i es capaz de asumir ese riesgo.

Una vez finalizada esta evaluación se realiza una nueva búsqueda *A**, ignorando aquellas celdas descartadas y disminuyendo el rango de $lookahead$ según la cantidad de celdas descartadas, luego de ello se obtiene un nuevo camino con las restricciones aplicadas.

3.2.2.3 Proceso de observación.

El proceso de observación consiste en que si se pierde de vista un agente j por parte de un agente i, entonces se eliminan los registros de este, ya que no se ven entre ellos. En caso de encontrarse ambos i y j a la vista, el agente i evalúa el comportamiento del agente j en base a las decisiones planeadas por j, es decir, el agente i espera que el agente j realice su próximo movimiento sobre la celda siguiente dentro del camino ideal de j. Esta predicción realizada será empleada dentro de la siguiente iteración del algoritmo para determinar los costos de conflicto.

3.2.2.3 Desplazamiento.

La última etapa consiste en realizar un movimiento a la siguiente celda prevista para el agente i, en caso de no ser posible entonces el agente evalúa regresar su celda anterior para evitar una potencial situación de interbloqueo. En caso de que el agente alcance su celda de destino este permanece inmóvil sobre ella y permite el paso de los demás agentes.

3.3 Modificaciones.

A continuación, se detallan aquellas modificaciones realizadas sobre cada implementación. Adicionalmente se empleó control de versiones *Git*[5] con el objetivo de mantener el correcto orden y seguimiento del trabajo realizado [6].

3.3.1 Modificaciones sobre ambas implementaciones.

Las primeras modificaciones se realizaron respecto a potenciales errores previstos por el compilador *GCC*, para ello se redujo la cantidad de alertas durante el proceso de compilación para ambas implementaciones, evitando de esta forma futuras falencias sobre la ejecución.

¿Eran errores o w

Sobre ambas implementaciones se añadió la funcionalidad que, al momento que un agente llegase a su celda de destino este se quedase estático y permitiese el desplazamiento de los demás agentes por sobre él. Respecto a los escenarios se adoptó el formato de lectura del mapeado desde un fichero externo (extensión *.map2*) y junto a cada mapa se vinculó un documento de localizaciones (extensión *.loc2*) con las ubicaciones de inicio y destino de cada agente. Adicionalmente al finalizar la ejecución de cada implementación se añadió la toma de muestras sobre un fichero externo para posteriormente determinar los resultados estadísticos con mayor facilidad.

¿Alguna justifica

3.3.1 Modificaciones sobre CR-MAPF.

Debido que, al escalar la cantidad de agentes, la implementación de *CR-MAPF* producía situaciones de interbloqueo ya que, de no avanzar un agente a la siguiente celda más probable dentro del camino calculado y de no poder retornar a su celda anterior, el agente se queda estancado sobre su celda actual. Para subsanar este problema se implementó una rutina nombrada como *push_out*. Esta rutina verifica la cantidad de ventanas de tiempo en las que un agente se queda inmóvil y en caso de exceder un límite previamente definido, el agente buscaría desplazarse a una celda libre aleatoria sobre sus celdas adyacentes. Cabe resaltar que el contador ventanas de tiempo para cada agente, se reinicia a 0 en caso de que el agente en cuestión logre desplazarse.

5. Git: Software de control de versiones, enfocado principalmente en el mantenimiento de aplicaciones con extensos archivos de código fuente.

3.4. Escenarios particulares.

3.4.1 Primera selección entre las celdas adyacentes

El primer escenario particular para destacar es la influencia que tiene la primera celda a ser insertada sobre el *heap* [10], esto durante el proceso *RTAA** de cada agente. Para ello consideremos el siguiente mapa (figura 3.1), sobre el cual se ejecuta *CR-MAPF* con un $lookahead$ de 3. En este escenario los agentes en color *azul* se desplazan a sus objetivos en color *naranjo*, cruzando por las celdas *blancas* y respetando los muros en color *verde*.

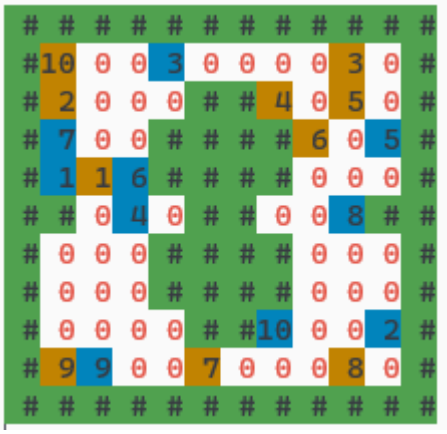


Figura 3.1: mapa test11, con 10 agentes

Se conoce que durante el proceso de *RTAA** se debe evaluar la siguiente celda que minimice el costo sobre el camino local en dirección la celda de destino, por ello la celda que queda en el *top* del *heap*, cumple con dos condiciones.

1. La siguiente celda minimiza costo del camino localmente.
2. La celda seleccionada fue la primera entre las celdas adyacentes al agente que minimizan el costo localmente.

La primera sentencia es evidentemente cierta dado el proceso *A**, mientras que la segunda hace referencia a los distintos posibles caminos *A**, los cuales localmente son factibles pero su elección es un proceso estocástico.

En tanto a la implementación la primera celda a evaluar se obtiene de forma aleatoria entre las celdas adyacentes. Esta celda evaluada se insertará sobre el *heap* y en caso de que luego de evaluar cada dirección esta resulte localmente óptima, entonces será

expandida, pero de llegar otra celda que sea igualmente óptima que la primera, entonces será la primera en ser expandida y no la nueva celda que acaba de ser evaluada.

Experimentalmente se observa lo siguiente (figuras 3.2 y 3.4), el agente 10 posee dos rutas igualmente factibles para su uso, ambas minimizan su coste (15 movimientos), pero la que elija dependerá en primera instancia de que celda sea insertada primero sobre el *heap*.

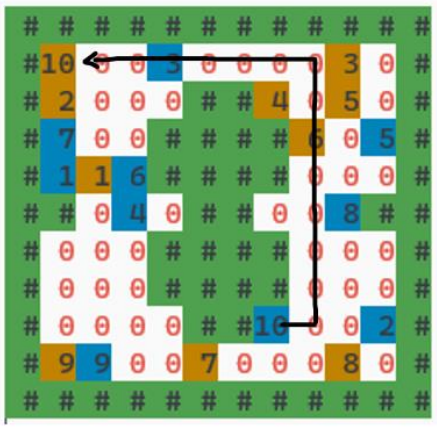


Figura 3.2: Posible ruta del agente 10, con costo de 15 movimientos

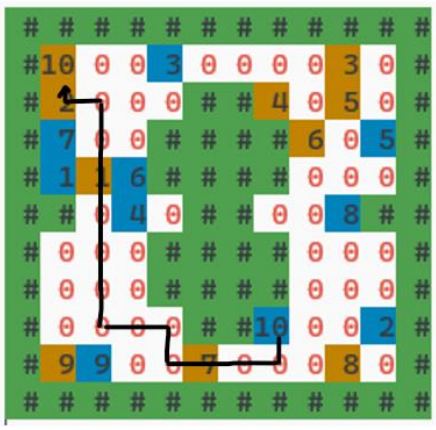


Figura 3.4: Posible ruta del agente 10, con costo de 15 movimientos

Es posible señalar que la evaluación sea desde un sentido en primera instancia y no aleatoria, lo cual favorecería a que el proceso fuese determinístico, pero esto llevaría a influenciar las decisiones tomadas por cada agente, sesgando los resultados, dado que ciertas direcciones tendrían prioridad por sobre otras.

3.4.2 Influencia del parámetro lookahead.

El parámetro *lookahead*, permite establecer el rango sobre el que se realiza la búsqueda *A**. Debido a la ejecución iterativa del algoritmo, establecer un valor alto sobre de $lookahead$ permite al agente visualizar más elementos de su entorno (ya sean muros u otros agentes), de modo el agente logra expandir un nodo que resulte más beneficio si es este el caso. Consideremos la figura 3.5, sobre la cual el agente indicado por un "1" en azul, debe desplazarse a su destino marcado por un "1" en naranjo.

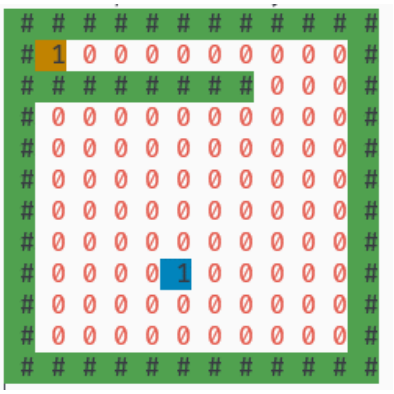


Figura 3.5: Escenario de prueba

Si sobre este escenario se ejecuta *RTAA** con un *lookahead* de 10 es posible obtener un resultado similar al de la figura 3.6, mientras que si se establece un *lookahead* de 100 el agente se desplaza sin problemas al destino realizando una búsqueda *A** completa, como es de apreciar en la figura 3.7.

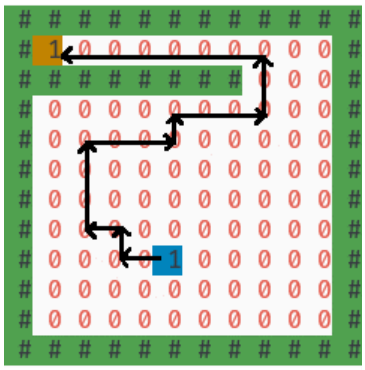


Figura 3.6: Escenario de prueba. lookahead=10

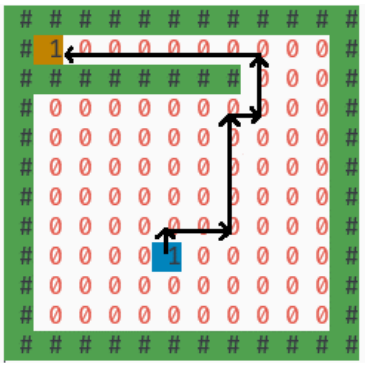


Figura 3.7: Escenario de prueba. lookahead=100

No hay manera en la que se pueda llegar de la situación presentada en la figura 3.8 a la presentada en la figura 3.9.

Bajo este escenario es posible llegar a acotar que mientras mayor es el *lookahead* mejor desempeño se obtendrá en relación con la cantidad de pasos y las iteraciones ejecutadas, pero esto no es cierto del todo, dado que al sobreestimar el valor del *lookahead,* los caminos determinados por cada agente presente en el escenario se solaparan en una etapa más temprana, lo que llevara a potenciales situaciones de interbloqueo entre los agentes. Para evidenciar este caso se considera el escenario de la figura 3.8, donde los agentes 1,2 y 3 en azul, buscan llegar a su objetivo 1, 2 y 3 en naranjo.

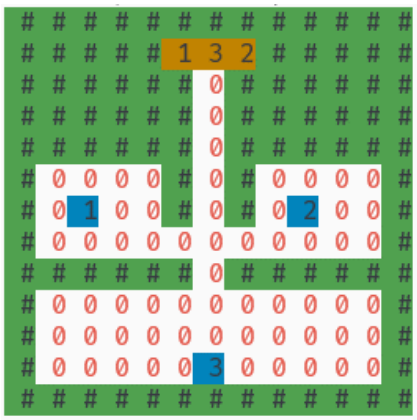


Figura 3.8: Escenario con múltiples agentes.

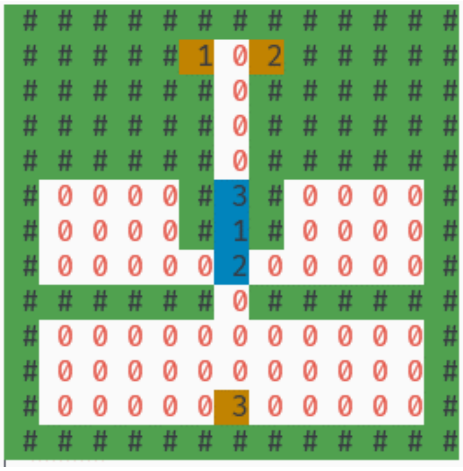


Figura 3.9: Escenario con múltiples agentes. Lookahead de 100

Con un $lookahead$ de 100, luego de 4 iteraciones se obtiene un escenario como el de la figura 3.9, donde los agentes 3 y 1 no se permiten el paso y el agente 2 queda circulando, intentando sobrepasar el bloqueo. Luego para el caso de utilizar un lookahead de 1, resulta que todos los agentes logran alcanzar su objetivo, pese a que los agentes 1 y 2 preseten un comportamiento errático en esa situación. Esto último se debe a la escaza información que maneja cada agente en un comienzo, permitiendo el paso al agente 3, por los pasos en falsos dados por los agentes 1 y 2.

Lo antes mencionado nos indica que el valor de $lookahead$ debe ser seleccionado bajo ciertos parametros adicionales, como lo son la distribución del escenario y la cantiadad de agentes presentes. Es de comprender que el algoritmo *RTAA** no entregue un resultado para cada situación, esto debido a su propia naturaleza de funcionar en tiempo de ejecución, lo que conlleva a problemas como los aquí presentados.

3.4.3 Influencia del valor formula.

Respecto a los efectos del valor formula sobre los agentes en el entorno durante la ejecución de *CR-MAPF* con un $lookahead$ de 3 se observa lo siguiente. Sobre la figura 3.10 se observan los agentes 1 y 2 en color azul, con objetivo las celdas 1 y 2 en color naranjo respectivamente.

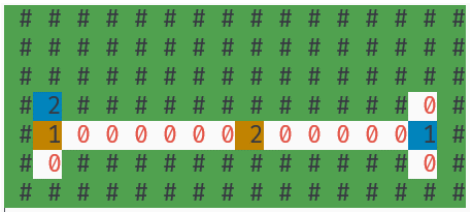


Figura 3.10: Escenario de prueba formula.

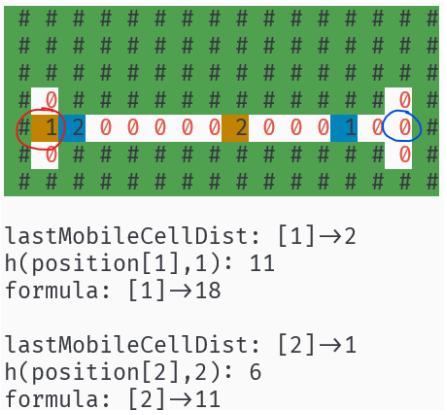


Figura 3.11: Escenario de prueba, dos etapas ejecutadas, distancia a celda y formula.

Sobre la figura 3.11 se observa el escenario luego de dos ventanas de tiempo, la celda encerrada en azul corresponde a la última celda que permite el movimiento en más de dos direcciones para el agente 1, mientras que la celda encerrada en rojo es para el agente 2. Por este motivo para el agente 1 la distancia a esta celda es de 2, mientras que para el agente 2 es de 1. En esta etapa la cantidad de celdas al objetivo para 1 es de 11 y para 2 es de 6. Esto lleva a que el resultado de $formula$ para cada agente sea de 18 y 11.

Dado que el $lookahead$ es 3, la figura 3.12 presenta el instante en el que ambos agentes logran visualizarse por primera vez, dado que el agente 1 posee una mayor formula y el agente 2 se encuentra sobre el camino del agente 1, entonces la situación de conflicto será favorable para el agente 1.

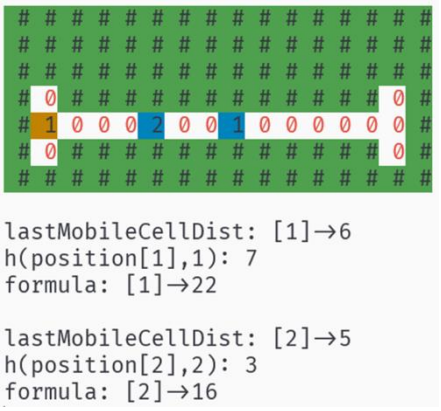


Figura 3.12: Escenario de prueba, etapa de visión mutua, distancia a celda y formula.

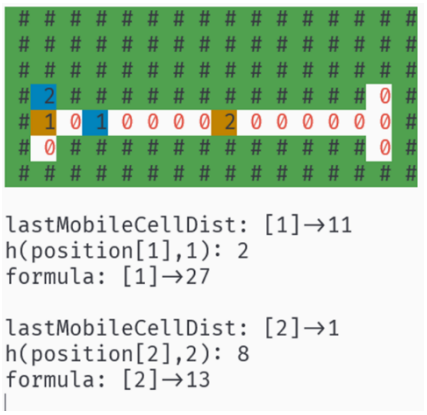


Figura 3.12: Escenario de prueba, agente 2 cede espacio, distancia a celda y formula.

Producto de las evaluaciones realizadas, el agente 2 acaba cediendo las celdas que había recorrido, ya que significa un coste menor que este recorra nuevamente esas celdas, comparado con la cantidad de celdas ya recorridas por el agente 1.

De esta manera la evaluación de $formula$ en esta situación se incrementa pese a que el agente 1 se aproxima a su celda objetivo, dado que esta depende en mayor medida de la ponderación respecto a la última celda que permitió al agente desplazarse en dos direcciones.

Capítulo 4.

EXPERIMENTACIÓN Y RESULTADOS.

En el presente capítulo se detallan las mediciones realizadas [7] sobre tres aspectos relevantes de las implementaciones propuestas. El primero de estos es la cantidad de agentes que logran conseguir llegar a su destino utilizando o no la nueva función *push_out* (únicamente sobre *CR-MAPF*). El segundo es comparar el desempeño en escalabilidad de ambas implementaciones (*Baseline* y *CR-MAPF*), sobre escenarios más extensos y con cantidades de agentes mayores. Mientras que el último aspecto a evaluar es la influencia del parámetro *lookahead* sobre un escenario y número de agentes definido.

4.1 Procedimiento de experimentación.

Para el procedimiento de experimentación se seleccionaron tres mapas con una escala de 34x34 celdas, estos fueron *libre*, *pasillo* y *4pasillo* (Anexo 1). Sobre todas las mediciones realizadas se utilizaron ubicaciones aleatorias de inicio y destino para cada agente, pero las mismas ubicaciones sobre el escenario comparativo. Para agilizar la tarea de asignar ubicaciones se diseñó un script, así también para facilitar la síntesis de la toma de muestras (ambos escritos en *Python*).

4.1.1 Variables y parámetros a considerar.

Las principales variables de entrada son:

- Mapa.
- Ubicaciones de cada agente.
- *lookahead*: Cantidad de celdas a expandir sobre *RTAA*.*
- *NAGENTS*: Cantidad de agentes sobre el mapa.

Se considera una *RUN* como la ejecución completa del algoritmo a utilizar, hasta el momento que todos los agentes logren llegar a su objetivo o se exceda el límite de iteraciones permitidas. Con esto en cuenta los parámetros a medir son:

- *sum_of_costs*: Cantidad promedio de movimientos realizada por los agentes durante la ejecución de una *RUN,* considerando la totalidad de agentes, independiente si este llega a destino o no.

- *makespan*: Cantidad promedio de ventanas de tiempo requeridas por agente durante la ejecución de una *RUN, en las que se consideran únicamente aquellos agentes que logran llegar a su destino*.
- *tiempo_ultimo_agente_goal*: Ultimo agente en llegar a su destino antes de acabar la ejecución de una *RUN*.
- *tiempo_en_acabar*: Tiempo requerido para acabar la ejecución completa de una *RUN* (menor o igual al límite de iteraciones permitidas).

Respecto a la implementación de *CR-MAPF*, se añaden los siguientes parámetros:

- *bad_pred*: Predicciones erradas
- *good_pred*: Predicciones acertadas
- *rate_pred*: Taza de acierto por predicción.
- *push_out_count*: Cantidad de llamadas a la función *push_out*.

Entre los aspectos que se conservarán inalterables son: las dimensiones del mapa, las iteraciones máximas permitidas en 1000, la cantidad de *RUNS* en 40 y la cantidad de iteraciones requeridas para hacer uso de la función *push_out* en 2, lo cual significa que, sobre una tercera iteración en una misma celda, el agente buscará desplazarse a una celda adyacente.

Previo a mostrar los resultados obtenidos es necesario considerar que, sobre aquellas ejecuciones en las que no se logre llegar con la totalidad de agentes a su celda de destino no es posible realizar una comparación exacta respecto a los parámetros *sum_of_costs* y *makespan* de cada método. Por ello, los resultados que se encuentren bajo esta condición serán utilizados únicamente para inferir comportamiento y no para determinar que método resulta más efectivo.

4.1.2 Arquitectura empleada.

En lo que respecta a la arquitectura empleada (Anexo 8) para la toma de mediciones y diversas pruebas realizadas, se utilizó una *CPU Intel Core i5-4430 @ 3.20GHz*, con *16GB* de *RAM* y una unidad *SSD* de *256GB*.

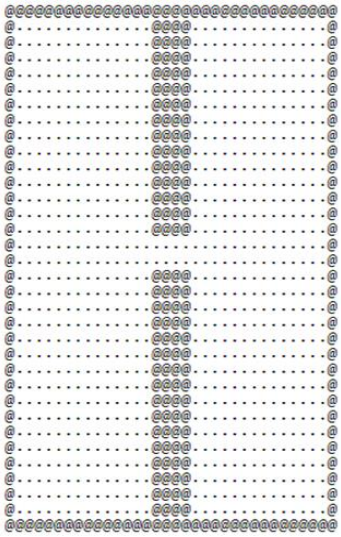
A nivel de software, el sistema operativo utilizado fue *Ubuntu 16.04.12*, el compilador *GCC 5.4.0* haciendo uso de la versión estándar de *C99*, el comando *make*, control de versiones *Git* y *Python 3.5.2*.

Al momento de realizar la experimentación ¿habían más procesos ejecutándose (además de los necesarios pa

4.2 Mediciones CR-MAPF / CR-MAPF-push_out.

Para medir el desempeño de la implementación *push_out*, se utilizó el escenario *pasillo*, con tramos de 10 agentes por muestra, hasta los 100 agentes en total. Se optó por un límite de 100 agentes y no superior dada la baja cantidad de agentes que lograban llegar a su destino en la implementación *CR-MAPF*.

La muestra en base a 40 ejecuciones por tramo de agentes (Anexo 2) exhibe un mejor comportamiento para *CR-MAPF-push_out* desde el tramo de 30 agentes. Mientras que para *CR-MAPF* se generan conflictos que no permiten completar la totalidad de agentes en su destino (figura 4.1).



Esquema 1: mapa pasillo.

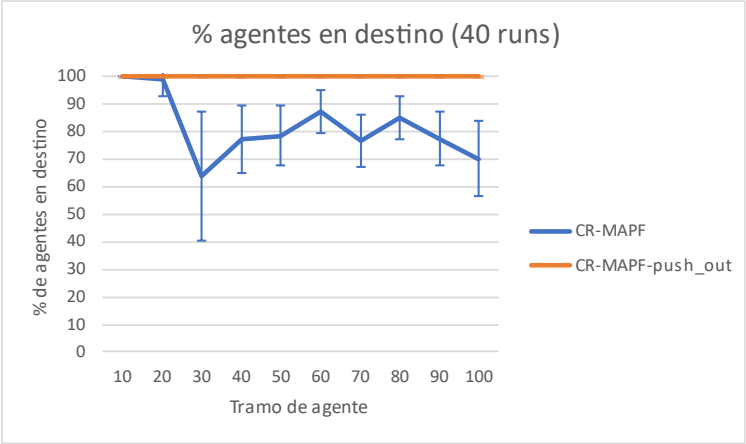


Figura 4.1: Pocentaje de agentes en destino.
Escenarios con y sin push_out.

Con este resultado es necesario considerar que el método *push_out* introduce aleatoriedad al momento de seleccionar una celda adyacente, por lo que la cantidad de veces que se llama este procedimiento debe ser la menor posible. En base a lo anterior para *CR-MAPF-push_out*, la comparación entre la totalidad de movimientos realizados con los movimientos dados por *push_out* es la siguiente.

¿Está bien la etic

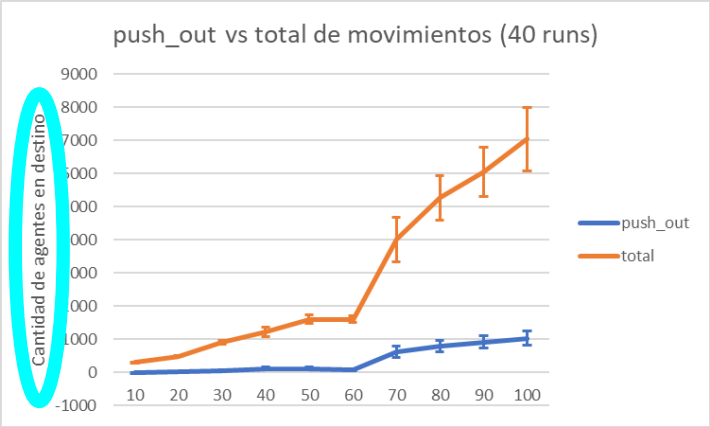
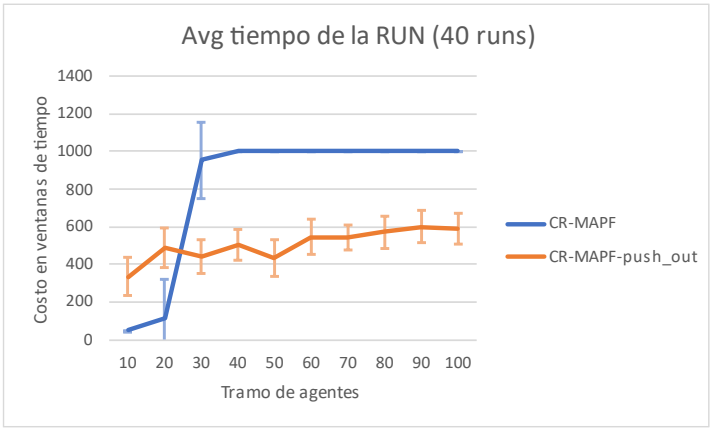


Figura 4.2: total de movimientos vs movimientos aleatorios.

En la figura 4.2 se muestra la cantidad de llamadas realizadas a *push_out*, las cuales no significan una gran parte de la totalidad de movimientos realizados. Precisando este resultado, para el tramo de 100 agentes se realizan en promedio *7035,4* movimientos, de los cuales *1023,175* son a raíz de un movimiento aleatorizado. Esto significa que en promedio solo un *14.54%* de los movimientos realizados sobre 100 agentes fueron aleatorios.



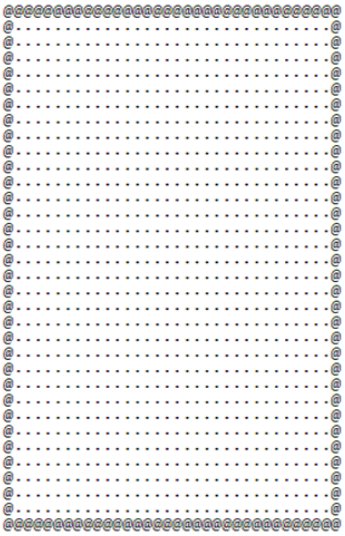
Me pregunto, ¿hay alguna manera d

Figura 4.3: Tiempo promedio requerido por RUN.

Como último aspecto, la figura 4.3 describe el tiempo requerido por cada método. La gráfica resume el promedio entre los tiempos máximos empleados por cada algoritmo y dado que *CR-MAPF* no logra llegar con la totalidad de agentes a sus celdas de destino, requiere de todo el tiempo permitido de la ejecución para acabar (situación de interbloqueo), lo cual no sucede sobre *CR-MAPF-push_out*.

4.3 Mediciones de escalabilidad CR-MAPF / Baseline.

Para las siguientes mediciones realizadas se consideraron un máximo de 200 agentes en simultáneo, tomando muestras por cada tramo de 10 agentes. Las implementaciones por utilizarse son *Baseline* y *CR-MAPF* con la rutina *push_out* habilitada. El valor de *lookahead* para la toma de muestras es de 3 sobre ambas implementaciones.



Esquema 2: mapa libre.

4.3.1 Mediciones en mapa libre.

Como resultado de la toma de muestras, para ambas implementaciones se consiguió que la completitud de los agentes llegase a su destino, esto por cada tramo en cada una de las 40 *RUNS* ejecutadas (Anexo 3).

Con lo anterior en consideración, en la figura 4.5 y figura 4.6 se observa un comportamiento similar en ambos escenarios. *CR-MAPF* presenta un resultado levemente mejor en la cantidad de movimientos realizados (*sum_of_costs*), el cual no llega a ser realmente significativo. Mientras que *Baseline* resulta utilizar una ~~pequeña~~ menor cantidad de ventanas de tiempo (*makespan*), al momento de aumentar la cantidad de agentes, lo cual de igual forma no entrega un resultado realmente significante.

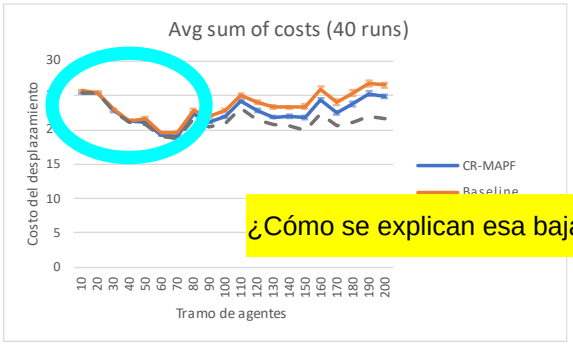


Figura 4.5: sum of costs, mapa libre, CR-MAPF vs Baseline.

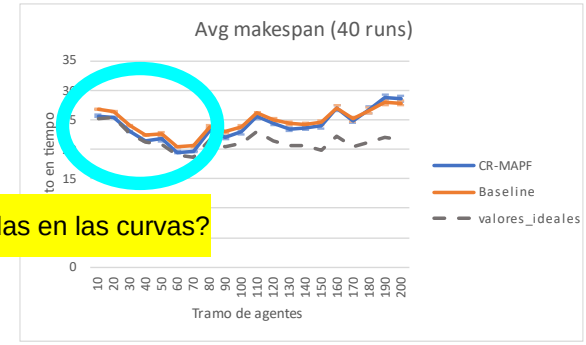


Figura 4.6: makespan, mapa libre, CR-MAPF vs Baseline.

Respecto a los parámetros específicos de *CR-MAPF* se observa lo siguiente. En la figura 4.7 se aprecia que las llamadas a *push_out*, son significativamente menores al total de movimientos realizados (en promedio el *1,44%* con 200 agentes). Lo anterior se explica

a raíz de la disposición del mapa, ya que, al no poseer muros, los agentes recorren fácilmente su camino y se reduce la cantidad de interbloqueos entre agentes.

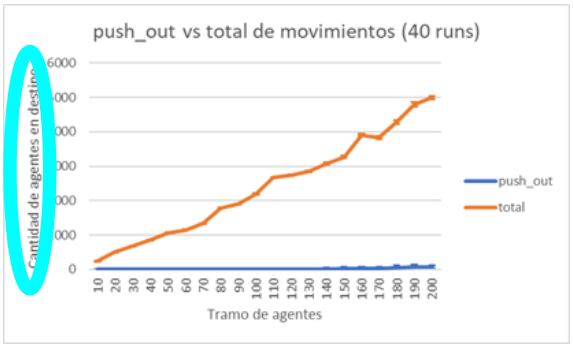


Figura 4.7: push_out, mapa libre, CR-MAPF.

Pese a que el mapa es bastante libre, la cantidad de predicciones erradas es alta (figura 4.8). Esto se debe a que, predicciones como seguir a otro agente o de intersección sobre una celda entre agentes tienden a errar, ya que en un mapa abierto los agentes habitualmente siguen su camino sin problemas y la cantidad de restricciones determinadas es menor.

Revisar etiqu

Figura 4.8: predicciones, mapa libre, CR-MAPF.

4.3.2 Mediciones en mapa pasillo.

Las mediciones en el mapa *pasillo* resultaron relativamente adversas (Anexo 4), esto dado que *Baseline* no logro llegar con la totalidad de sus agentes en escalas superiores a 100 (figura 4.9). Pese a esto es posible analizar su comportamiento hasta 100 agentes.

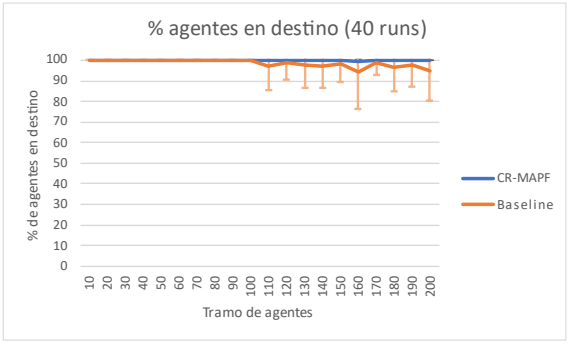
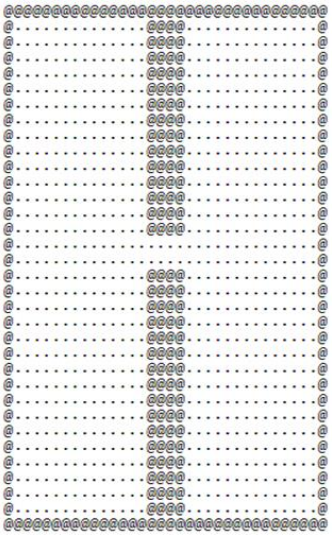


Figura 4.9: Promedio de agentes en destino, mapa pasillo.



Esquema 3: mapa pasillo.

Con lo anterior en consideración los resultados respecto al *sum_of_costs*, muestran un mejor desempeño hasta los 150 agentes (figura 4.10), donde luego se iguala con *Baseline* y pasa a tener un desempeño peor, pese a ello, la dispersión de los datos se dispara en el caso de *Baseline*, para tramos por sobre los 100 agentes. El mismo escenario se repite en el caso de *makespan*, donde luego del tramo de 100 agentes *Baseline* muestra un mejor resultado, pero con una dispersión considerablemente mayor a la seguida en tramos anteriores (figura 4.11). Este problema se explica dado que al interbloquearse los agentes en *Baseline* y no alcanzar la totalidad de los agentes llegar a su destino, parte de la población de agentes queda estancada, reduciendo artificialmente la cantidad de movimientos realizados. Mientras que sobre el parámetro *makespan*, dado que no se logra llegar por cada agente a su destino, únicamente entrarían en esta grafica aquellos que si, por lo que, la figura 4.11 solo muestra una parte de los agentes sobre el tramo de 100 agentes.

Para *CR-MAPF* su desempeño fue bastante favorable, donde únicamente se produjo una perdida en el tramo de 160 agentes, pero pese a ello, alcanzando un promedio de 159,95 agentes y una desviación estándar de 0,3162, lo cual realmente no implica un problema relevante sobre las mediciones entregadas (de manera comparativa a *Baseline*).

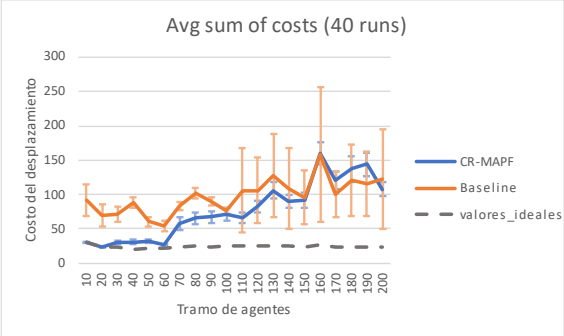


Figura 4.10: sum of costs, mapa pasillo, CR-MAPF vs Baseline.

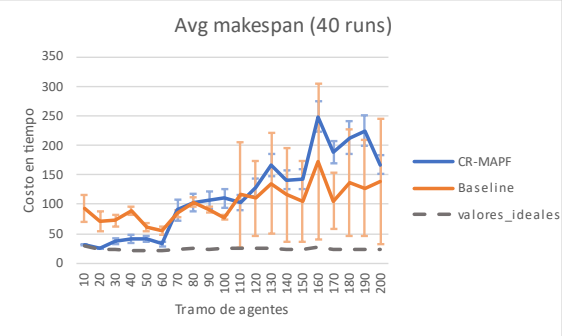


Figura 4.11: makespan, mapa pasillo, CR-MAPF vs Baseline.

Respecto a los parámetros específicos de *CR-MAPF* se observa lo siguiente. La cantidad de llamadas a *push_out* (figura 4.12) representa un *13,15%*, en el tramo de 200 agentes, lo cual es bajo y acorde con un buen funcionamiento.

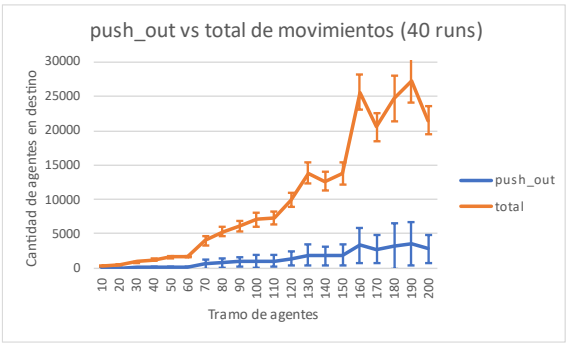


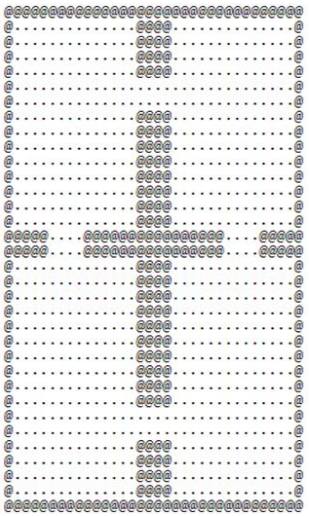
Figura 4.12: push_out, mapa pasillo, CR-MAPF.

Referente a las predicciones realizadas, se observa un incremento circunstancial de las predicciones acertadas por sobre el tramo de 110 agentes. Esto se debe principalmente a la densidad de agentes que se agolpa sobre el pasillo, ya que al momento de querer estos cruzar por el estrecho pasillo ubicado en el centro, los agentes con caminos similares tienden a realizar la predicción o ver de manera favorable, el seguir el camino que lleva el otro agente en conflicto, dado que se dirigen en un mismo sentido.

Figura 4.13: predicciones, mapa pasillo, MARL.

4.3.3 Mediciones en mapa 4pasillo.

Sobre el mapa *4pasillo* (Anexo 5), lograron llegar a su destino la totalidad de los agentes en el caso de *CR-MAPF* y con una pequeña pérdida para *Baseline* (figura 4.14), sobre los tramos de 170 y 190 agentes. Pese a que *Baseline* posee pérdida en los tramos 170 y 190, el promedio de agentes es de 168,375 y 188,975, con una dispersión de 7,18 y 6,48 respectivamente, lo cual no es del todo relevante como para ignorar estos resultados, aun así, dado que sobre los tramos de 180 y 200 agentes se logra la completitud de agentes en destino, se utilizará estos datos para describir el comportamiento a escala mayor.



Esquema 4: mapa 4pasillo.

¿Alguna intuición pa

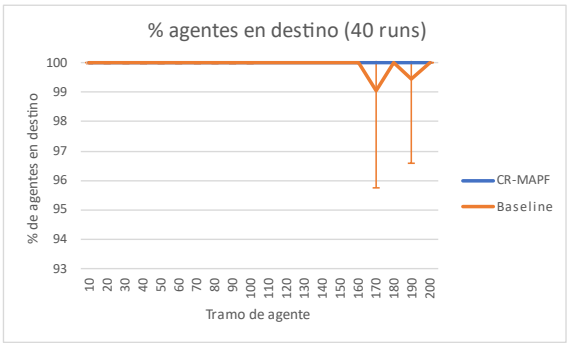


Figura 4.14: promedio de agentes en destino, mapa 4pasillo, CR-MAPF vs Baseline.

Analizando el *sum_of_costs* (figura 4.15), se observa que *CR-MAPF* posee un comportamiento bastante similar al ideal *A** a baja escala. Aumentando la escala *CR-MAPF* se iguala en desempeño con *Baseline*, desde el tramo de 130 a 160 agentes, esto se explica por la densidad de agentes en el mapa. Al aumentar la cantidad de agentes se incrementan las restricciones realizadas sobre los caminos de cada agente, lo que conlleva a que los agentes utilicen caminos levemente más extensos, a costa de incrementar el total de movimientos realizados.

Respecto al *makespan* (figura 4.16), la implementación de *CR-MAPF* demanda un mayor tiempo para acabar desde el tramo de 90 agentes, esto es de esperarse dado que al momento de determinar una restricción que lleve al agente a detenerse o interbloquearse este agente con otro agente en su camino, se pierden ventanas de tiempo en las que el agente se conserva estático sobre su casilla, lo cual se coindice con la cantidad de llamadas a *push_out* (figura 4.17).

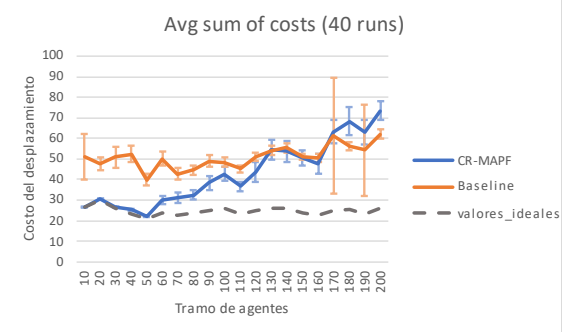


Figura 4.15: sum_of_costs, mapa 4pasillo, CR-MAPF vs Baseline.

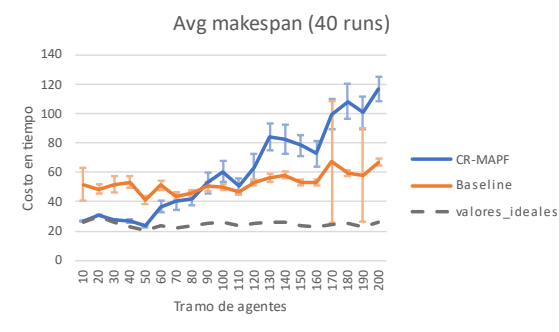


Figura 4.16: makespan, mapa 4pasillo, CR-MAPF vs Baseline.

Respecto a los parámetros específicos de *CR-MAPF* se observa lo siguiente. La cantidad de llamadas a *push_out* (figura 4.17) representa un *14,3%*, en el tramo de 200 agentes, lo cual es levemente superior en comparación con las muestras de *pasillo,* pero permite explicar el incremento respecto al parámetro *makespan*.

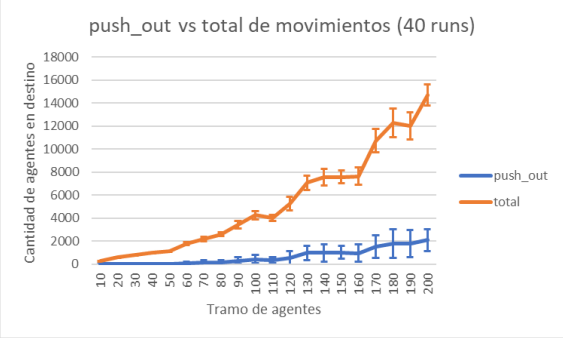


Figura 4.17: push_out, mapa 4pasillo, CR-MAPF.

En tanto a las predicciones (figura 4.18) no existe gran diferencia al mapa *pasillo*, principalmente dado que los agentes optan por seguir los otros agentes que cruzan por cada pasillo antes de quedarse estáticos.

Figura 4.18: predicciones, mapa 4pasillo, CR-MAPF.

4.4 Mediciones sobre lookahead.

Para la evaluación de la variable *lookahead* se utilizó el mapa pasillo (Anexo 6), sobre el tramo de 50 agentes, esto ya que en este tramo *CR-MAPF* y *Baseline* presentaban un buen desempeño y resulta ser la mitad del último tramo en el que todos los agentes en *Baseline* llegan a su destino, de esta manera se evita favorecer una implementación por sobre la otra. Los valores de *lookhead* a utilizar, escalan mediante tramos de 3, hasta

alcanzar los 30. También se considera que *CR-MAPF* se encuentra con su rutina *push_out* habilitada.

Con lo anterior en consideración, se observa una pérdida de agentes en *Baseline* para los tramos 15, 21, 24 y 27 (figura 4.19), aun así, los tramos 18 y 30 no presentan anomalías, por lo que se utilizarán estos últimos de forma comparativa con *CR-MAPF*.

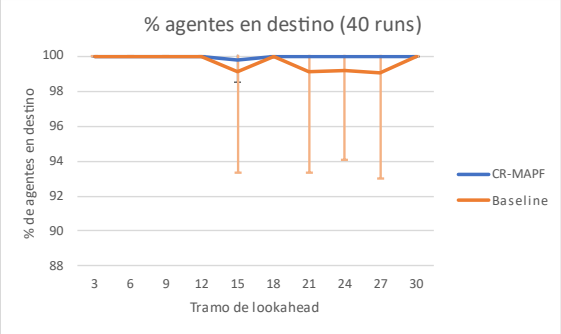
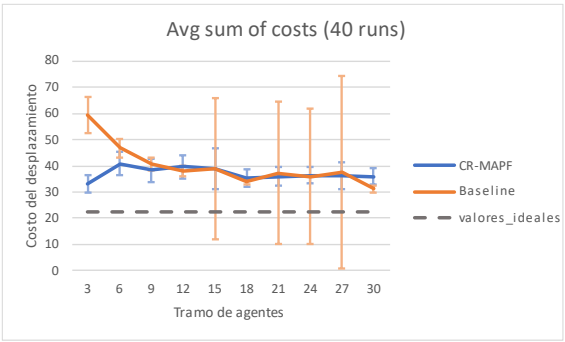


Figura 4.19: promedio de agentes en destino, mapa pasillo.

Respecto a la evolución del *sum_of_costs* (figura 4.20), este parámetro al verse incrementado el valor del lookahead, tiende conservar los resultados sobre *CR-MAPF,* sin alguna variación aparente, al contrario de *Baseline* el cual al incrementar este valor tiende a seguir caminos más consistentes con su camino óptimo a la celda de destino, lo cual lleva a reducir los movimientos realizados en los tramos 18 y 30.

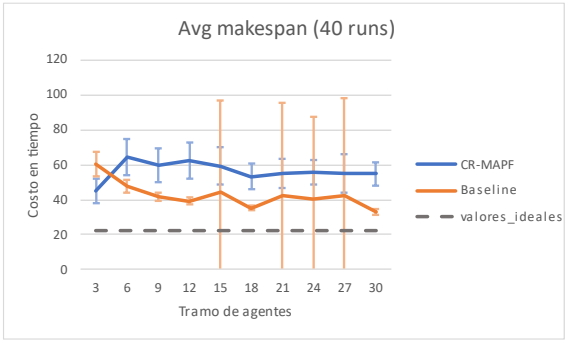


Si bien el baseline genera caminos más

Figura 4.20: sum of costs, mapa pasillo, CR-MAPF vs Baseline.

Sobre los resultados del parámetro *makespan* (figura 4.21), es de esperar que al incrementar el *lookahead* la información de búsqueda (celdas sobre el camino) sea mayor, lo que conlleva que para *CR-MAPF*, las restricciones determinadas de igual forma se incrementen. Pese a ello *CR-MAPF* no ve su desempeño significativamente mermado

respecto a *Baseline*, el cual, si bien le supera desde el tramo *lookahead* 6, no posee una tendencia a mejorar aún más su desempeño.



Lo mismo acá, la variabilidad de baselir

Figura 4.21: makespan, mapa pasillo, CR-MAPF vs Baseline.

Respecto a los parámetros específicos de *CR-MAPF* se observa lo siguiente: la cantidad de llamadas a *push_out* (figura 4.22) tiende a conservarse, al incrementar el *lookahead*. Esto es de esperarse dado que la cantidad de llamadas a esta función se incrementa con la cantidad de agentes en cuestión, debido a la alta densidad de estos sobre el mapa.

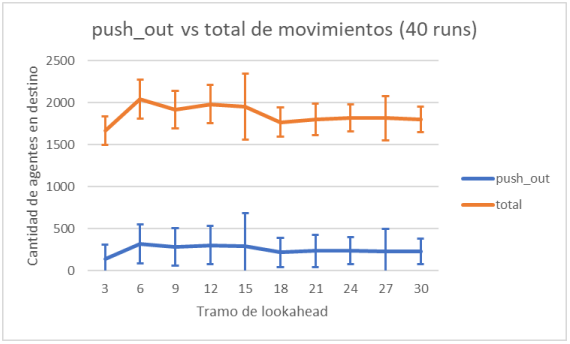


Figura 4.22: push_out, mapa pasillo, CR-MAPF.

Respecto a las predicciones realizadas (figura 4.23), estas tienden a incrementarse linealmente, dado que por cada tramo se incrementa igual cantidad de nuevas predicciones a realizarse, aun así, un valor alto de *lookahead*, no muestra favorecer a *CR-MAPF* en absoluto en esta situación.

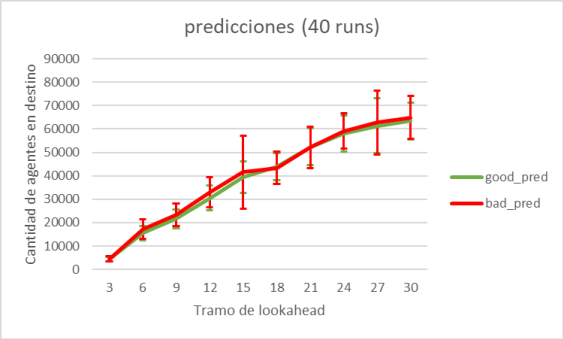


Figura 4.23: push_out, mapa pasillo, CR-MAPF.

4.5 Análisis generales.

Sobre los resultados anteriores es posible concluir que *CR-MAPF* presenta mejores resultados en mapas más restrictivos (como lo es *pasillo*), mientras que en mapas con mayor libertad de desplazamiento (como *4pasillo* y *libre*) el escenario resulta ser parcialmente favorable a *Baseline* (a escalas mayores), respecto al *sum_of_costs* y *makespan*. Esto último se debe principalmente a las restricciones aplicadas en *CR-MAPF*, en escenarios donde la congestión producto del movimiento de los agentes confluya a un único pasillo o a pasillos más estrechos, la cantidad de celdas de conflicto es mayor y los agentes se agolpan por intentar atravesar desde un extremo a otro. En el caso contrario si el mapa posee menos restricciones o pasillos más anchos (*4pasillo* posee pasillos con ancho 4), *Baseline* consigue un mejor resultado dado que conserva siempre en movimiento a los agentes (ya que no evalúa celdas de conflicto) y al encontrarse un mapa con mayor libertad estos logran subsanar la congestión y llegar a su objetivo, en cambio, *CR-MAPF* determina restricciones, mantiene agentes estáticos bajo ciertas circunstancias y acaba por disminuir levemente su desempeño.

Respecto a qué algoritmo logra llegar con mayor cantidad de agentes a su celda de destino, *CR-MAPF* utilizando *push_out* presenta un mejor comportamiento respecto a *Baseline* en todas las mediciones aquí realizadas, por este motivo resulta más segura su utilización en caso de que se desee asegurar la mayor cantidad de agentes en destino.

¿Alguna explicación para explicar la alta variabilidad del base

Capítulo 5.

CONCLUSIONES Y RECOMENDACIONES.

Los resultados obtenidos en este estudio muestran un mejor comportamiento para la implementación *CR-MAPF con push_out* respecto a *Baseline en escenarios donde la interacción multiagente resulta mayor y mapas ~~donde~~ más restrictivos*. Esto ya sea en tramos bajos de agentes (donde logra reducir el *sum of costs* y el makespan) como en tramos altos (donde en la mayoría de las ejecuciones logra llegar con todos los agentes a su celda de destino). Aún en escenarios menos restrictivos como lo son *libre* y *4pasillo*, es necesario considerar que *CR-MAPF con push_out* no posee pérdida de agentes sobre ningún tramo, lo que lleva a concluir que pese a igualarse con *Baseline* (caso *libre*) o ser parcialmente más costoso en ventanas de tiempo (caso *4pasillo*), es más seguro utilizar *CR-MAPF con push_out* dado que sobre estos tramos logra la completitud de agentes en destino.

Adicionalmente existen consideraciones que se encuentran ciertamente ignoradas, la primera son las dimensiones del escenario en cuestión, dada la excesiva demanda en *CPU* para la implementación *CR-MAPF*, se optó por utilizar mapas de dimensiones menores, con una cantidad de agentes acotada. Este último problema no ocurre en *Baseline*, dado que al no restringir los caminos producto de *RTAA**, no demanda mayor consumo de *CPU.*

Otro aspecto importante del cual no se dio registro preciso, es el comportamiento de ambas implementaciones con un *lookahead* mayor y con mayor cantidad de agentes en el mapa. Respecto a este último punto existen dos problemas, si bien para *CR-MAPF*, el incrementar el valor de *lookahead* demanda un mayor consumo de *CPU*, la implementación *Baseline* muestra ciertas anomalías al escalar este valor, por lo cual de igual forma se espera que esta no logre acabar con todos sus agentes si se incrementa este valor junto con la cantidad de agentes.

¿Cuáles anom

De manera resolutoria, si bien *CR-MAPF* muestra un mejor desempeño, la implementación propuesta muestra falencias al momento de escalar la cantidad de agentes, lo cual en ciertos aspectos dificulta la toma de muestras y el proceso de experimentación. Aun así, es evidente que *CR-MAPF* resulta reducir los parámetros *sum of costs* y *makespan*, evitando una pérdida significativa de agentes y con una mejor adaptabilidad a

la diversidad de mapas, lo cual significa un mejor resultado respecto a *Baseline*. Pese a que la selección de mapas resulta beneficiosa para *Baseline* (por la reducida cantidad de muros) esto no provocó una seria desventaja para *CR-MAPF*, el cual sobre el mapa *pasillo* (que resulta ser el más restrictivo) reluce todas las ventajas de utilización.

Recomendaciones.

Sobre *CR-MAPF* a nivel de implementación, existen una gran variedad de ajustes (realizados y por realizar) que mejorarían significativamente el consumo de *CPU*, con esto en consideración es realmente importante optar por una máquina con una alta tasa de GHz en mono núcleo u optar por paralelismo para reducir los tiempos de ejecución. Otro aspecto importante es la toma de muestras, el trabajo realizado permite almacenar de manera distendida los datos producto de la ejecución de múltiples *RUNS*, lo cual facilita la manipulación de las métricas y la generación de resultados, reduciendo el tiempo requerido para la recolección de muestras. Adicionalmente es recomendable adoptar la utilización de los formatos *map2* y *loc2*, para evitar diseñar el mapa con sus ubicaciones sobre la implementación del algoritmo.

Trabajo futuro.

Respecto a la implementación de *CR-MAPF*, existen ajustes que deberían corregirse, los cuales pese a no modificar su desempeño provocan que la interpretación del código escrito sea dificultosa (ejemplo: Utilizar arreglos multidimensionales en vez de tipos de datos específicos en *C*). Continuando sobre la implementación, esta posee una cantidad excesiva de ciclos iterativos, si bien algunos de ellos fueron condensados sobre un mismo ciclo en común, existen otros que se conservan sin alterarse. Corregir este aspecto reduciría en parte el alto consumo de *CPU* requerido. Como último aspecto de la implementación a modificar, sería lograr utilizar mapas con dimensiones mayores y una mayor cantidad de agentes sobre estos, tomando en cuenta un escenario en que la alta demanda de *CPU* se vea reducida.

A nivel del algoritmo *CR-MAPF*, la implementación de la función *push_out*, resultó ser bastante efectiva para escalar la cantidad de agentes, aun así, este introduce una aleatoriedad en los resultados que no es posible ignorarse. En reemplazo a esta (o en complemento), sería recomendable utilizar la velocidad media de los agentes como valor de referencia para aquellos agentes que se encuentren en sus proximidades, por ejemplo, sobre las últimas celdas en las que un agente se desplazó, realizó tres movimientos y una

detención, dado que esta información es observable por los demás agentes (no existe coordinación explícita), se permitiría priorizar el movimiento de aquellos agentes que alcancen velocidades más altas, permitiendo conservar su momentum[6].

6. Momentum: En mecánica clásica, la cantidad de movimiento se define como el producto de la masa del cuerpo y su velocidad en un instante determinado.

Referencias.

[1] Jun-tao Li, Hong-jian Liu. Design Optimization of Amazon Robotics. Automation, Control and Intelligent Systems. https://www.semanticscholar.org/paper/Design-Optimization-of-Amazon-Robotics-Li-Liu/860301b748c4e154048bb0ca7b37eda5a67a08fb

[2] S. Koenig and M. Likhachev. Real-Time Adaptive A*. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 281-288, 2006. http://idm-lab.org/bib/abstracts/papers/aamas06.pdf

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.). MIT Press and McGraw–Hill. pp. 595–601. ISBN 0-262-03293-7.

[4] Russell, Stuart J. (2018). Artificial intelligence a modern approach. Norvig, Peter (4th ed.). Boston: Pearson. ISBN 978-0134610993. OCLC 1021874142.

[5] Parra (2012). Microprocesadores. RED TERCER MILENIO S.C.

http://www.aliat.org.mx/BibliotecasDigitales/sistemas/Microprocesadores.pdf

[6] Manuel Díaz, Repositorio MT-CR-MAPF, con las implementaciones de Baseline y CR-MAPF, https://github.com/AmigoManuel/MT-CR-MAPF/

[7] Manuel Díaz, Toma de muestras y mediciones, con las implementaciones de Baseline y CR-MAPF, https://github.com/AmigoManuel/MT-CR-MAPF/tree/main/memoria

[8] Devon Sigurdson, Vadim Bulitko, William Yeoh, Carlos Hernandez and Sven Koenig, Multi-Agent Pathfinding with Real-Time Heuristic Search, http://idm-lab.org/bib/abstracts/papers/cig18.pdf

[10] Manuel Díaz, Registros en video respecto al funcionamiento, https://1drv.ms/u/s!ArbVB3pAebsnjL9eD3m44Quf8LxTnA?e=F58D3V

[11] Richard Johnsonbaugh, Discrete Mathematics. Capítulo 8. 8th Edition, ISBN-0321964683.

Referencias imágenes.

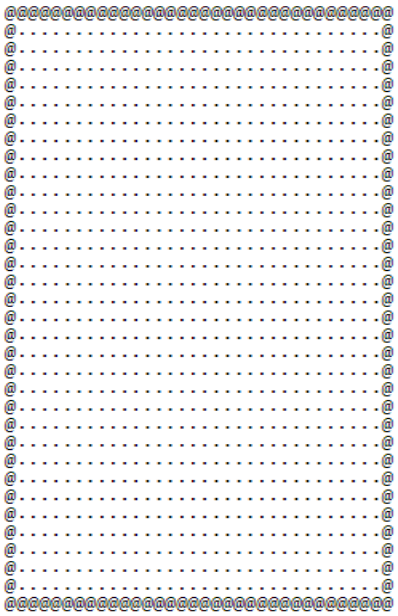
[9] Introduction to A* From Amit’s Thoughts on Pathfinding, https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Anexos.

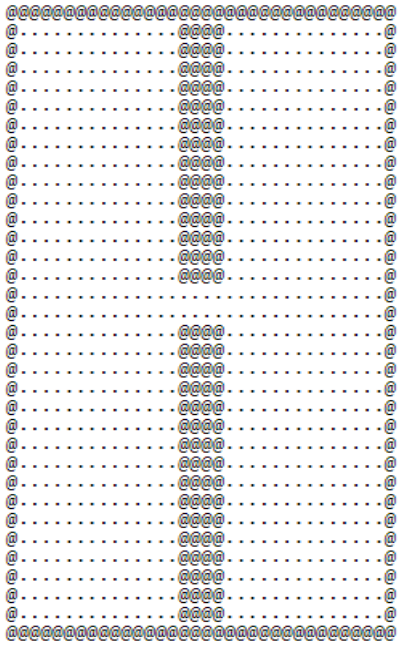
Anexo 1.

Mapas seleccionados para experimentación.

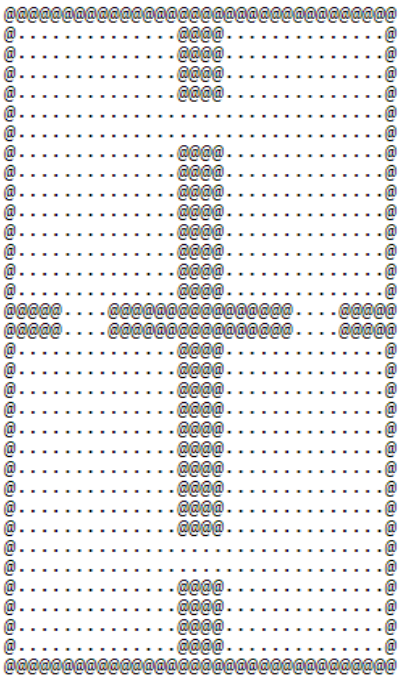
- *libre*. Es un mapa de 34x34 celdas, que permite la libertad de desplazamiento para los agentes.



- *pasillo*. Mapa de 34x34 celdas, con un estrecho túnel en su centro.

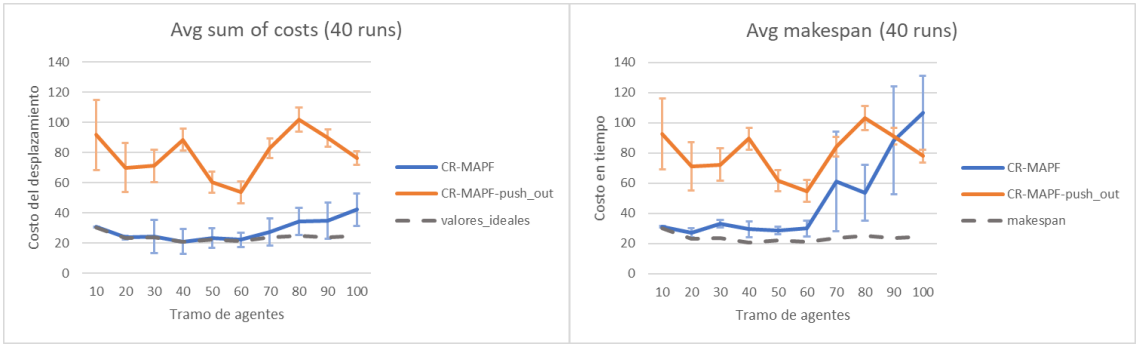


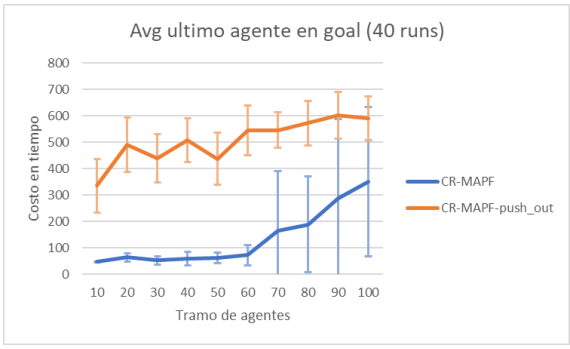
- *4pasillo*. Mapa de 34x34 celdas, con 4 túneles estrechos.

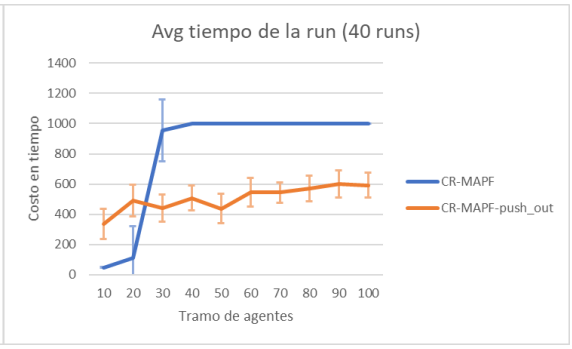


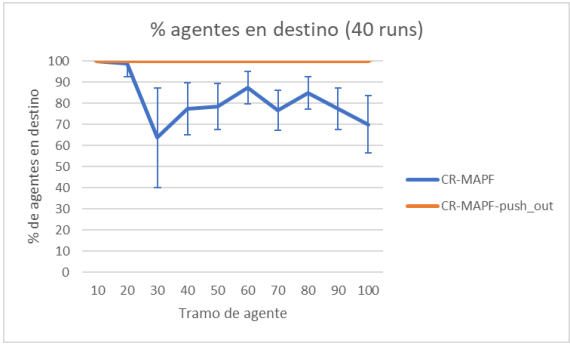
Anexo 2.

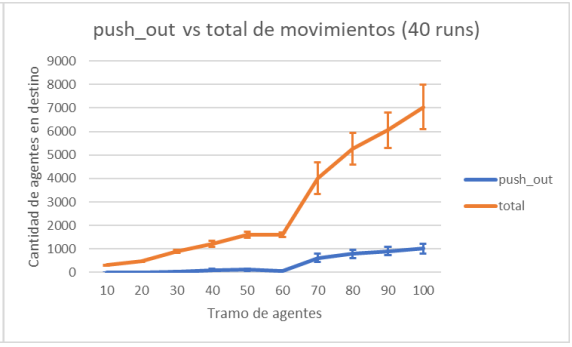
Mediciones CR-MAPF vs CR-MAPF-push_over, con un máximo de 100 agentes y un incremento de 10 agentes por muestra.





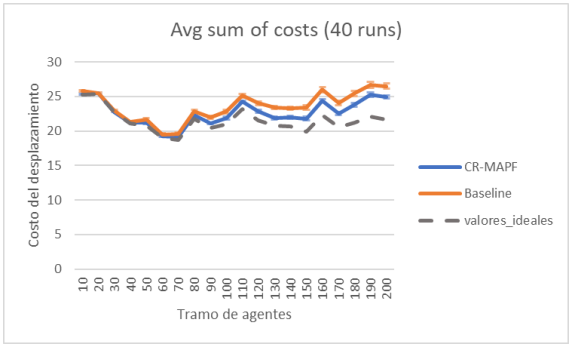


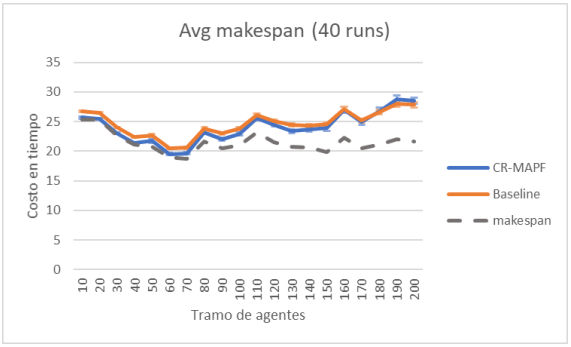


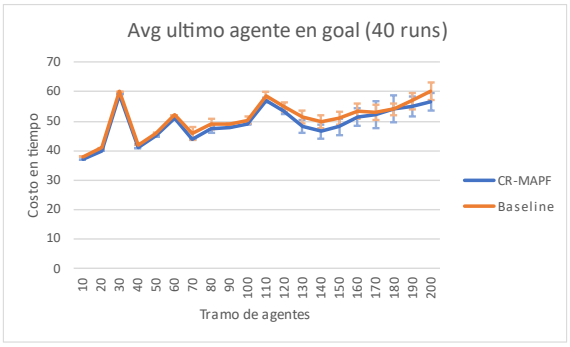


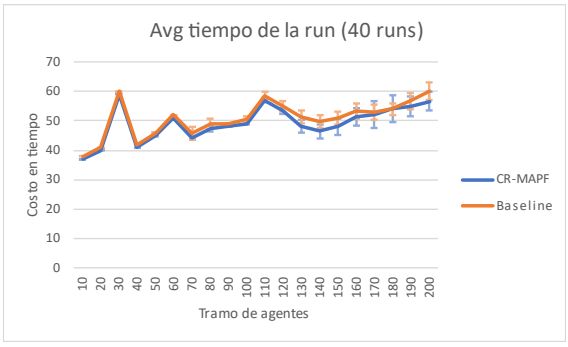
Anexo 3.

Mediciones de escalabilidad, sobre mapa *libre*, con un máximo de 200 agentes y un incremento de 10 agentes por muestra.

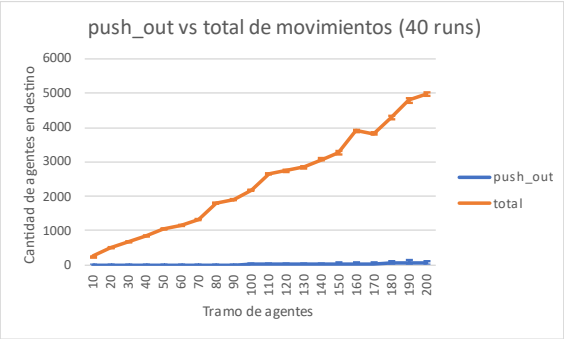


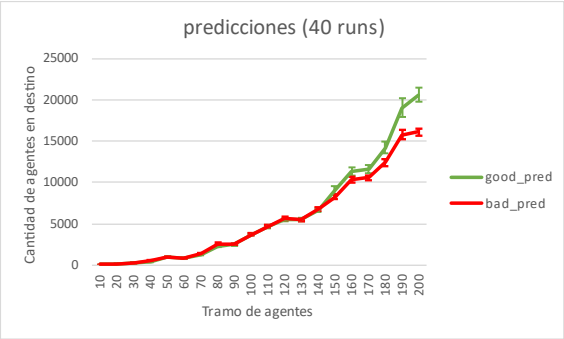


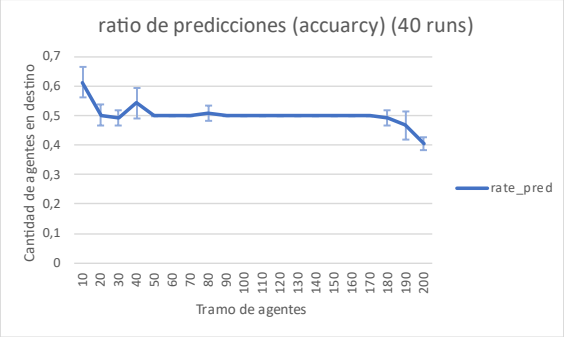




Mediciones aspectos de *CR-MAPF*.

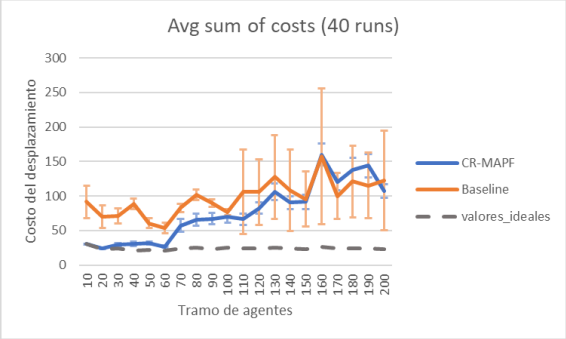


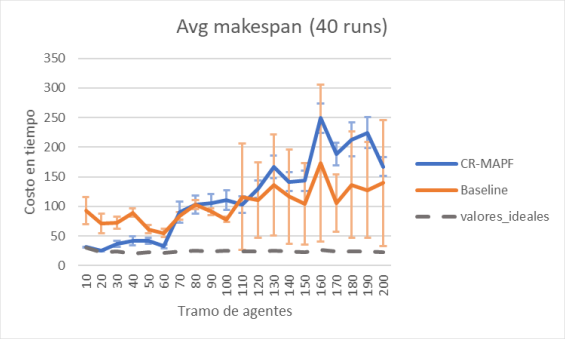


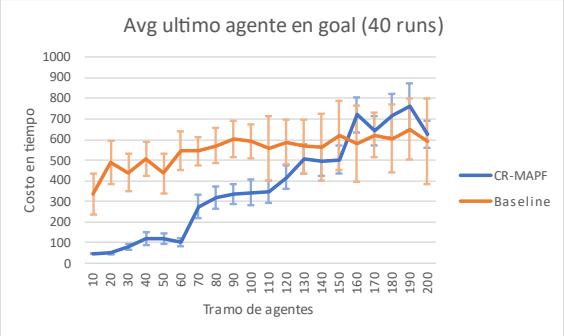


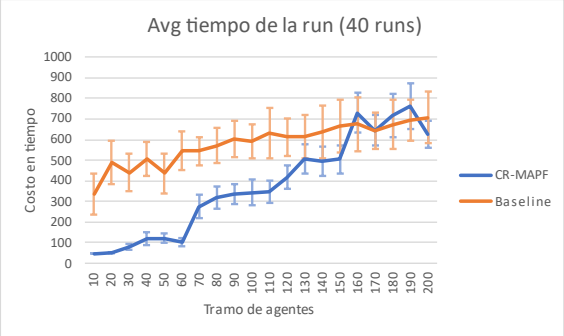
Anexo 4.

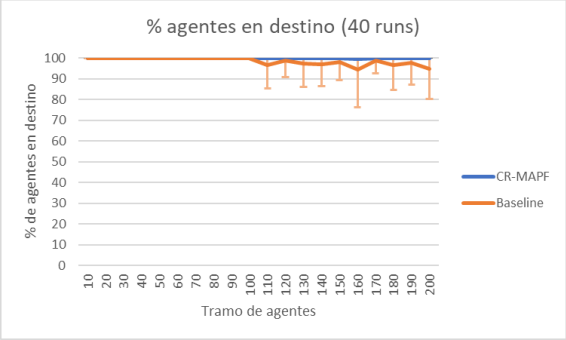
Mediciones de escalabilidad, mapa *pasillo*, con un máximo de 200 agentes y un incremento de 10 agentes por muestra.



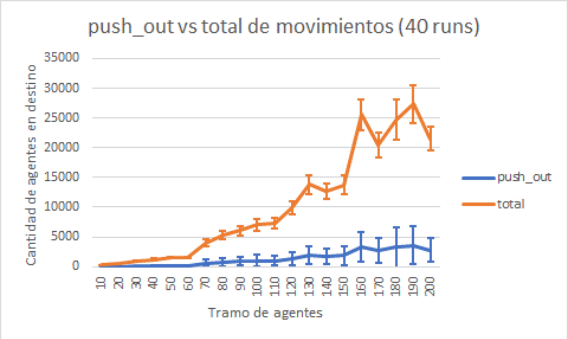


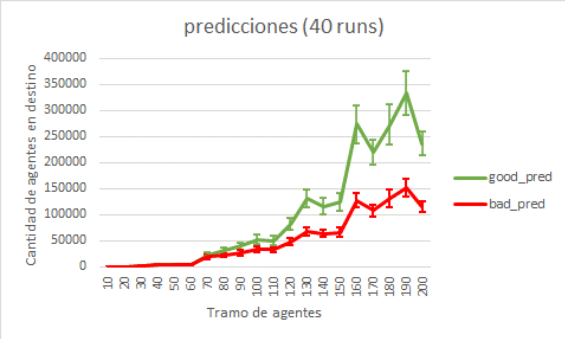


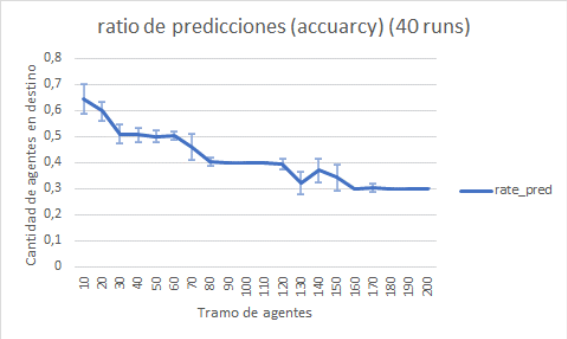




Mediciones aspectos de *CR-MAPF*.

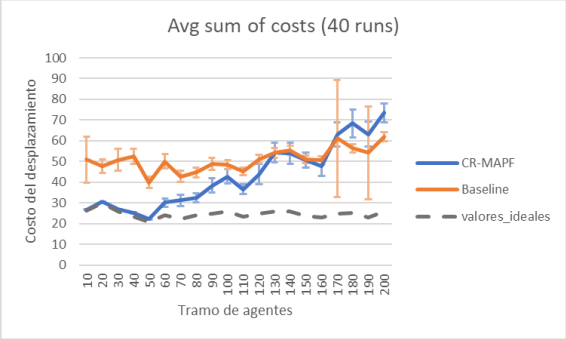


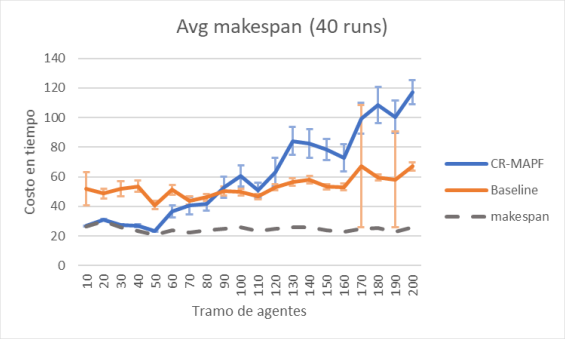


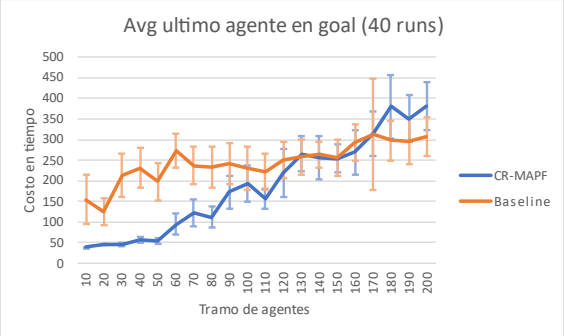


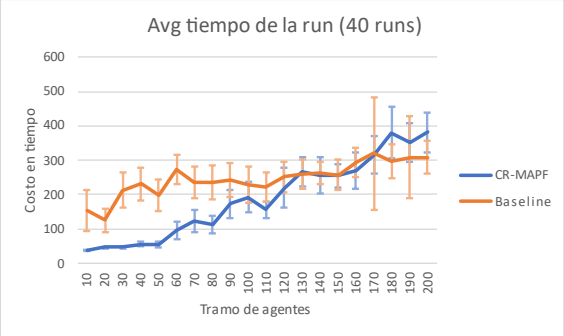
Anexo 5.

Mediciones de escalabilidad, mapa *4pasillo*, con un máximo de 200 agentes y un incremento de 10 agentes por muestra.

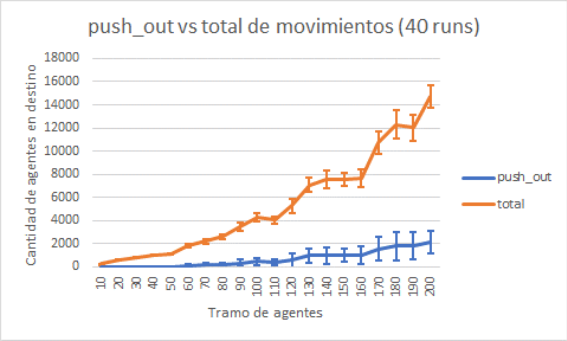


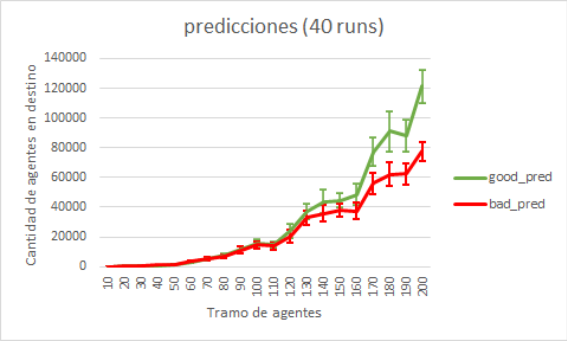


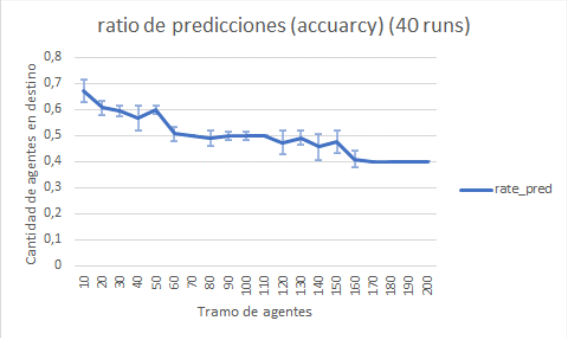




Mediciones aspectos de *CR-MAPF*.

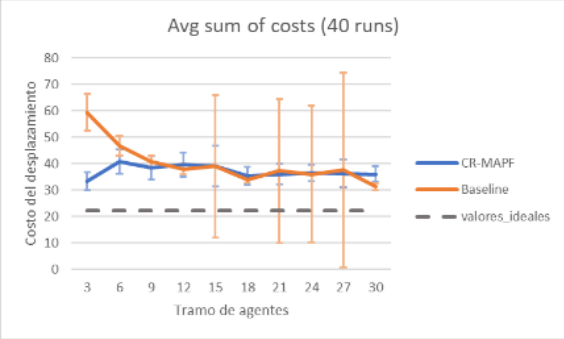


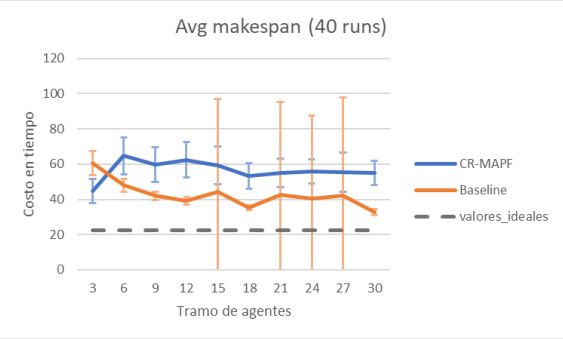


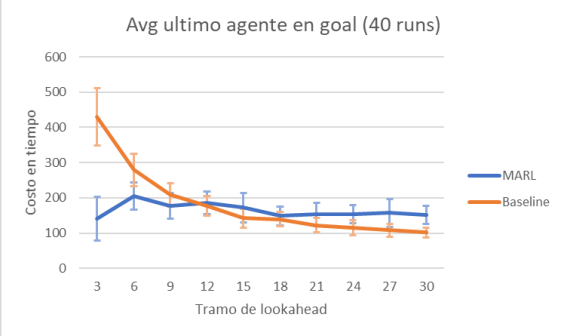


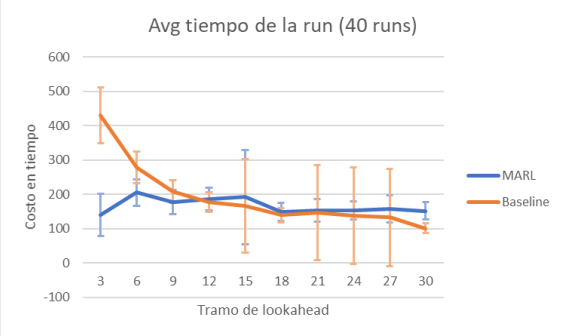
Anexo 6.

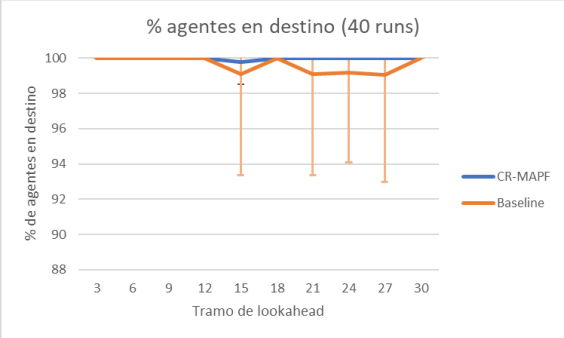
Mediciones *lookahead*, con un máximo de 30 estados/celdas con un incremento de 3 estados/celdas por muestra.



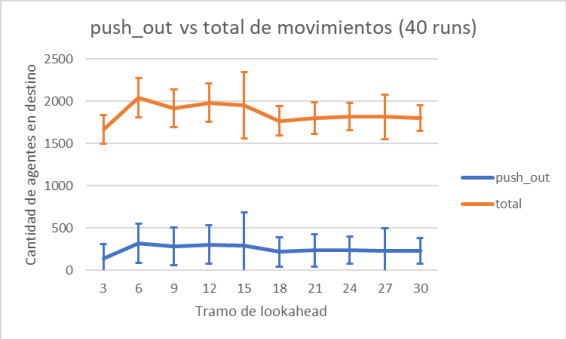


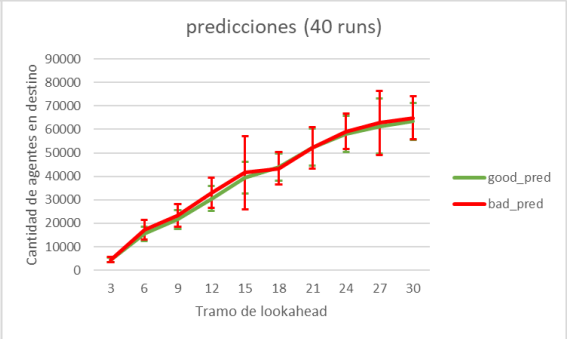


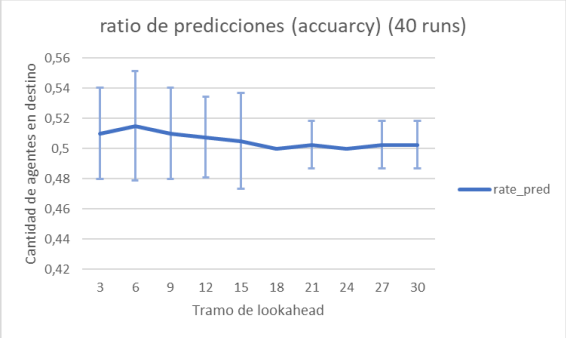




Mediciones aspectos de *CR-MAPF*.







Anexo 7.

Algoritmo detallado CR-MAPF. El algoritmo que sigue CR-MAPF es el siguiente.

Algorithm 4: CR-MAPF

```
1  newrandommaze_astar()
2  initialize_astar()
3  for cada agente i do
4      for cada agente j do
5          if j ≠ i then
6              // Primera observación, determina si los agentes i y j se ven entre ellos
7              observe_agent(j, i, lookahead, position[i])
8  while no llegue cada agente a su destino do
9      for cada agente i do
10         if el agente no se encuentra en su destino then
11             // Determina el camino de i (ignorando colisiones) y los costos de conflictos
12             compute_shortestpath_astar(i, lookahead)
13             for cada instante l dentro del lookahead do
14                 if se encuentra determinado ideal_path para i en el instante l then
15                     if l es posterior al inicial y existe un ideal_path para l-1 then
16                         if conflict_cost[i][ideal_path[i][l-1]][l-1] es mayor a 0.5 then
17                             transformar este en 1 para recalcar esta dificultad
18                     se asigna el valor de l como el largo del camino para el agente i
19             // Segunda observación, aplica las restricciones mediante conflict_cost[i]
20             if compute_constraintpath(i, lookahead) then
21                 if la celda de destino se encuentra bloqueada then
22                     conserva su posición
23                     pasa al siguiente agente
24                 else
25                     asigna nueva posición a i
26                     for cada agente i do
27                         for cada agente j do
28                             // Realiza nueva observación
29                             observe_agent2(j, i, lookahead, nueva celda de i)
30                             observe_new_agents(i, j, lookahead)
31             else
32                 // No encontró solución, debe regresar por su camino
33                 if celda anterior a i se encuentra bloqueada then
34                     conserva su posición y pasa al siguiente agente
35                 else
36                     cambia la celda actual del agente i por su celda anterior
37             if agente i llego a su casilla de destino then
38                 se queda en el lugar permitiendo el paso por sobre el
39             if cada agente llego a su casilla de destino then
40                 acaba la ejecución
```

En primera instancia *CR-MAPF* realiza por cada agente una observación sobre los demás agentes mediante $observe\_agent(j, i, lookahead, position[i])$, determinando si estos se encuentran dentro del rango de visión. Este método inicializa el cálculo de las predicciones y por cada agente incrementa los contadores $blockedAgent$, $fromTransition$, $toTransition$, $agentMovingTo$ en 1, sobre $position[i]$, esto por cada agente i y j que logren verse mutuamente una cantidad de z, donde z se incrementa desde el 0 hasta el valor de $lookahead$.

Estos contadores significan:

- $position[i].blockedAgent[j][z]$: j observa que la celda $position[i]$ se encuentra bloqueada por i en el instante z.
- $position[i].fromTransition[j][z]$: j observa que hay un desplazamiento desde la celda $position[i]$ por parte de i en el instante z.
- $position[i].toTransition[j][z]$: j observa que hay un desplazamiento hacia la celda $position[i]$ por parte de i en el instante z.
- $position[i].agentMovingTo[j][z][i]$: j observa a i desplazarse a $position[i]$ en el instante z

Cada uno de contadores antes mencionados serán esenciales en el proceso de determinar las restricciones. Ahora dado que en primera instancia no se ha realizado movimiento alguno, es correcto afirmar que cada agente ve a sus vecinos estáticos sobre su celda inicial, conservando de esta forma los contadores en $position[i]$.

CR-MAPF / compute_shortestpath_astar.

Una vez finalizada la primera observación, se continua con la ejecución del ciclo $while$ (línea 8 del algoritmo CR-MAPF), este queda en ejecución mientras queden agentes por llegar a su celda de destino. Una vez dentro del ciclo iterativo, se evalúa por cada agente i que no se encuentre en su celda de destino, el camino mínimo desde su celda actual hasta el objetivo, mediante la ejecución de $compute\_shortestpath\_astar(i, lookahead)$. Esta última función construye el $path[i]$ desde la celda actual de i en dirección a su celda de destino, con un máximo de $lookahead$ celdas e ignorando las posibles colisiones, pero, realizando la evaluación conflictos y determinando el valor $formula$, el cual permite conocer la prioridad que debe tener un agente dado que su movimiento se ve desfavorecido por la disposición de sus celdas adyacentes.

Antes de determinar el valor $formula$ se determina la ultima celda que permitió el movimiento del agente i en más de dos direcciones y se almacena como $lastMobileCell[i]$. Una vez hecho esto el valor de $formula$ se encuentra dado por.

$$\boldsymbol{formula = h(position[i], i) + ((2 * distancia(lastMobileCell[i])) + 3)}$$

Donde $distancia()$ evalúa la distancia desde la posición actual $position[i]$ hasta la celda $lastMobileCell[i]$ y $h()$ entrega la distancia a la celda objetivo desde la celda $position[i]$ para i. Este valor de $formula$ permitirá a aquellos agentes que la maximicen capitalizar de mejor manera las celdas al momento de evaluar las restricciones, el considerar la ponderación $2 * distancia(lasMobileCell[i])$ permite a aquellos agentes que se trasladen a través de pasillos estrechos poseer una $formula$ mayor.

El algoritmo $compute\_shortestpath\_astar$ es el siguiente.

Algorithm 5: CR-MAPF / compute_shortestpath_astar(int a, int lookahead)

```
1  // Determina lastMobileCellDist
2  lastMobileCell[a] ← ultima celda en la que a puede desplazarse en más de dos direcciones
3  formula ← h(position[a], a) + ((2 * distance(lastMobileCell[a])) + 3)
4  agentInfo[a] ← formula
5  // Ahora ejecuta A* desde la posición actual de a
6  mazestart1 ← position[a]
7  mazegoal1 ← goal[a]
8  insertheap2(mazestart1)
9  cont_closed ← 0
10 while !heap2.empty() or cont_closed > lookahead do
11     tmpcell ← heap2.pop()
12     cont_closed ← cont_closed + 1
13     // Determina restricciones en la primera expansión
14     if cont_closed == 1 then
15         determine_constraints(a, lookahead, formula, tmpcell, 1)
16     // Evalúa celdas adyacentes
17     for cada celda adyacente a position[a] do
18         // Verifica si adyacente no ha sido visitada antes, Ajuste A*
19         if adyacente.g ≥ tmpcell.g + tmpcell.cost[adyacente] then
20             adyacente.g ← tmpcell.g + tmpcell.cost[adyacente]
21             // Se determinan restricciones de esta expansión
22             determine_constraints(a, lookahead, formula, tmpcell.move[adyacente], 0)
23             heap2.push(adyacente)
24 // reconstruye el camino mediante las celdas expandidas
25 path[a] ← camino a estrella de el agente a
```

El valor de $formula$ es almacenado por agente en $agentInfo$ y es actualizado cada vez que se llama a esta función.

Luego de esto se ejecuta el proceso *A** desde la celda actual del agente i en dirección a su celda de destino, con un máximo de $lookahead$ celdas por determinar. Una vez finalizado el proceso, se reconstruye el camino siguiendo las celdas expandidas y es almacenado en $path[i]$.

El proceso para la determinación de conflictos es ejecutado una vez por cada celda expandida, mediante $determine\_constraints(i, lookahead, formula, tmpcell, initialState)$, donde $initialState$ indica si corresponde a la primera celda en el camino o no.

CR-MAPF / determine_role.

Previo a detallar la evaluación de conflictos es necesario conocer el proceso para determinar el rol a tomar por la celda $determine\_rol$. Esta función compara la formula entregada por dos agentes i y j, donde si el agente con mayor $formula$ no es i, entonces se debe determinar $conflict\_cost$, ya que, sobre la celda en conflicto, i tiene menor $formula$ que j. El algoritmo es el siguiente.

Algorithm 7: CR-MAPF / determine_role(int roleij, int maxInfo, int a, int j, int cell_role)

```
// roleij igual a 1 respeta el movimiento de a, roleij igual a 0 el de otro agente
if roleij no ha sido definido then
    if maxInfo == a then
        roleij ← 1
    else
        roleij ← 0
// Se debe determinar conflic_cost
if roleij == 0 then
    cell_role ← 0
// Esta celda esta libre de conflicto para a
if roleij == 1 and cell_role≠0 then
    cell_role ← 1
```

Si $role[i][j]$ no ha sido determinado y el agente con mayor formula es i, entonces $role[i][j] = 1$, en caso contrario $role[i][j] = 0$. Ahora si $role[i][j] == 0$ es porque hay un agente j con mayor formula y se debe determinar el $conflict\_cost$, en caso de que $role[i][j] == 1$ la celda se encuentra libre de conflicto.

CR-MAPF / determine_constraints.

Respecto a *determine_constraints*, su algoritmo es el siguiente.

Algorithm 6: CR-MAPF / determine_constraints(a,lookahead,formula,currentCell,initialState)

```
1  numConflicts ← 0 // contador de conflictos
2  maxInfo ← 0 // agente con mayor formula
3  cell_role ← -1 // como se debe tratar a currentCell
4  for future ← pathlength[a] to lookahead by 1 do
5      if currentCell.blockedAgent[a][future] or currentCell.blockedAgent[a][future - 1] then
6          // caso 1 - se evalúa ultimo instante en el camino de a
7          if future == pathlength[a] then
8              for cada agente j do
9                  if j ≠ a and currentCell.agentMovingTo[a][future][j] then
10                     if j esta en su destino then
11                         Continuar con el siguiente agente
12                     numConflicts ← numConflicts + 1
13                     if formula < agentInfo[j] then
14                         maxInfo ← j
15                     determine_role(role[a][j], maxInfo, a, j, cell_role)
16         // caso 2 - Se espera que currentCell se encuentre bloqueada por a en future - 1
17         if currentCell.blockedAgent[a][future - 1] and future > 0 then
18             for cada agente j do
19                 if currentCell.agentMovingTo[a][future - 1][j] then
20                     numConflicts ← numConflicts + 1
21                     if formula < agentInfo[j] then
22                         maxInfo ← j
23                     determine_role(role[a][j], maxInfo, a, j, cell_role)
24         // caso 3 - hay un agente en actualCell
25         if currentCell.blockedAgent[a][0] then
26             for cada agente j do
27                 if currentCell == position[j] then
28                     numConflicts ← numConflicts + 1
29                     maxInfo ← j
30                     determine_role(role[a][j], maxInfo, a, j, cell_role)
31         if no existen conflictos, numConflicts == 0 then
32             for cada agente j do
33                 if currentCell.agentMovingTo[a][future - 1][j] then
34                     if j esta en su destino then
35                         Continuar con el siguiente agente
36                     numConflicts ← numConflicts + 1
37                     if formula < agentInfo[j] then
38                         maxInfo ← j
39                     else
40                         maxInfo ← a
41                     determine_role(role[a][j], maxInfo, a, j, cell_role)
42         if cell_role == 0 then
43             step ← 0
44             if initialState == 0 then
45                 step ← pathlength[a]
46             if conflict_cost[a][currentCell][step] == 0 then
47                 if ocurre caso 2 then
48                     conflict_cost[a][currentCell][future] = 1 / ((future - 1) - step + 2)
49                 else
50                     conflict_cost[a][currentCell][step] = 1 / (future - step + 2)
```

En primera instancia se inicializa un contador de conflictos $numConflicts$, un indicador del agente que contenga la mayor formula $maxInfo$ y un indicador del cómo se debe tratar la $currentCell$ al momento de evaluar los conflictos.

Esta función evalúa tres escenarios, considerando que $currentCell$ es una celda candidata a formar parte del camino ideal para el agente i y $future$ itera desde el valor de $pathlength[i]$ hasta $lookahead$.

El primer escenario es cuando existe otro agente j, no en su celda de destino, que quiera acceder a $currentCell$ sobre el instante $future == pathlength[i]$ ($currentCell.agentMovingTo[i][future][j]$), de ser así, se incrementa $numConflicts$ y si $formula$ de j es mayor que la actual, se toma a j como el nuevo agente de referencia y se determina el rol de la celda, mediante $determine\_role(role[i][j], maxInfo, i, j, cell\_role)$. En este caso es necesario recordar que $pathlength[i]$ se incrementa debido a la cantidad de celdas expandidas, pese a que de momento el $path[i]$ no se encuentre determinado, $pathlength[i]$ si se conoce y este comienza desde 0 hasta un máximo de $lookahead$.

El segundo escenario considera que $currentCell$ se encuentre bloqueada un instante antes de $future$ ($currentCell.blockedAgent[a][future - 1]$), de ser así se busca el agente j que desea desplazarse a $currentCell$ en $future - 1$ ($currentCell.AgentMovingTo[a][future - 1][j]$), al encontrarse este se incrementa $numConflicts$ y si $formula$ de j es mayor que la actual, se toma a j como el nuevo agente de referencia. Una vez acabado este escenario se determina el rol de la celda, mediante $determine\_role(role[i][j], maxInfo, i, j, cell\_role)$.

El tercer escenario ocurre cuando hay un agente sobre $currentCell$ en el instante actual ($currentCell.blockedAgent[a][0]$), de ser así se busca el agente j que mantiene bloqueada $currentCell$ ($position[j] == currentCell$), se incrementa el contador de conflictos, se toma a j como el nuevo agente de referencia. Una vez acabado este escenario se determina el rol de la celda, mediante $determine\_role(role[i][j], maxInfo, i, j, cell\_role)$.

Si luego de esta evaluación no existen conflictos, se realiza una última búsqueda. Si existe un agente j, no sobre su destino, que desee desplazarse a $currentCell$ en el instante anterior a $future$ ($currentCell.agentMovingTo[a][future - 1][j]$), entonces se incrementa $numConflicts$, si $formula$ del agente j es mayor que la actual entonces se toma a j como

el nuevo agente de referencia, en caso contrario se toma a como el agente de referencia y se determina el rol de la celda mediante $determine\_role(role[i][j], maxInfo, i, j, cell\_role)$.

Para finalizar en caso de que $cell\_role == 0$ entonces se debe determinar el $conflict\_cost$, si la determinación de conflictos es sobre la primera celda del camino *A** ($initialState == 1$) entonces se toma el instante $step = 0$, si se trata de otra celda entonces se utiliza el instante $step = pathlength[i]$. En la situación que el $conflict\_cost$ de i en $currentCell$ sobre el instante $state$ no ha sido inicializado ($conflict\_cost[i][currentCell][state] == 0$), existen dos escenarios:

- En la situación que el caso 2 se haya dado, entonces quiere decir que existe un agente sobre $currentCell$ al momento de i intentar desplazarse y el $conflict\_cost$ esta dado por $conflict\_cost[i][currentCell][future] = 1/((future - 1) - step + 2)$
- Si el caso 2 no ocurrió, entonces significa que otro agente intenta desplazarse a $currentCell$ y el $conflict\_cost$ se encuentra dado por $conflict\_cost[i][currentCell][state] = 1/(future - step + 2)$.

Una vez finalizado el proceso ya se contará con $conflictCost$ determinado, el cual será utilizado al momento de aplicar las restricciones sobre el camino.

CR-MAPF / compute_constraintpath.

Una vez finalizado el proceso $compute\_shortestpath\_astar$ ya se cuenta con las situaciones de conflicto evaluadas en $conflictCost$, por lo cual para determinar el nuevo camino con restricciones a seguir es necesario guiarse con los valores entregados por $conflictCost$. El algoritmo por utilizar es el siguiente.

Algorithm 8: CR-MAPF / compute_constraintpath(int a, int lookahead)

```
1  limitcell ← pathlength[a]
2  l ← pathlength[a]
3  while l ≥ 0 do
4      if conflictCost[a][path[a][l]][l] > 0.49 then
5          path[a][l].marked[a][l] = bloqueada
6          limitcell = limitcell - 1
7      if pathlength[a] > l and limitcell == l and 0.01 < conflictCost[a][path[a][l]][l] ≤ 0.49
        and path[a][l].grado[a] < 3 then
8          if l ≠ 0 then
9              path[a][l].marked[a][l] ← bloqueada
10             limitcell ← l - 1
11         else
12             path[a][l].marked[a][l] ← disponible
13             path[a][l].marked[a][l + 1] ← bloqueada
14             limitcell ← l
15     if pathlength[a] > l and limitcell == l and 0.201 < conflictCost[a][path[a][l]][l] ≤
        0.49 and path[a][l].grado[a] ≥ 3 then
16         if l ≠ 0 then
17             limitcell ← l - 1
18         else
19             limitcell ← l
20         path[a][l].marked[a][l] ← disponible
21         path[a][l].marked[a][l + 1] ← bloqueada
22     l ← l - 1
23 new_lookahead ← pathlength[a] - limitcell
24 if new_lookahead > 0 then
25     mazestart1 ← path[a][limitcell]
26     insertheap2(mazestart1)
27     for l to lookahead by 1 do
28         path[a][l] ← null
29     a* con objetivo goal[a] y lookahead ← new_lookahead
30     reconstruye el camino path[a]
```

$limitCell$ indica la celda limite sobre la cual las celdas siguientes a ella serán celdas con restricciones, inicialmente se toma $limitCell = pathlength[i]$ ya que idealmente se espera no tener que aplicar ninguna restricción y conservar este valor. El $grado[i]$ de una celda corresponde a la cantidad de celdas adyacentes libres para el agente i sobre esa celda.

Luego se itera desde la última celda sobre el camino ideal de i hasta alcanzar la primera, realizando las siguientes evaluaciones.

- Si el $conflictCost[a][celda][l] > 0.49$, entonces se descarta esa celda del camino, ya que significa un riesgo de conflicto muy alto.
- Si $0.01 < conflictCost[a][celda][l] \leq 0.49$, el $celda.grado[a] < 3$, la cantidad de celdas $limitcell$ coincide con el instante l y l es un instante previo al final, entonces a menos que fuese el estado inicial el agente no debiera acceder a esta celda, dado que restringe su movilidad, por ello si $l \neq 0$ se marca bloqueda, mientras que si $l = 0$ se permite le permite al agente acceder a ella.
- Si $0.201 < conflictCost[a][celda][l] \leq 0.49$, el $celda.grado[a] \geq 3$, la cantidad de celdas $limitcell$ coincide con el instante l y l es un instante previo al final, entonces si no es el estado inicial se decrementa $limitcell$, de ser el estado inicial se deja $limitcell = 1$ y se marca la celda como disponible, esto significa que probablemente exista un agente sobre esa celda, pero el riesgo es menor y el agente i desea tomar ese riesgo.

Una vez finalizada la evaluación de restricciones se contará con celdas que anteriormente se encontraban en $path[i]$ ahora marcadas a ignorarse. Se determina el $new\_lookahead$, se inicializa un nuevo $heap$ con la celda inicial $path[i][limitcell]$ y se realiza una nueva búsqueda *A**, sobre la cual se ignoran las celdas marcadas como bloqueadas y se restringe el $lookahead$ a $new\_lookahead$. Ya finalizado este proceso se cuenta con un nuevo camino $path[i]$ con las restricciones aplicadas.

CR-MAPF / observe_agent2 y observe_new_agents.

Previo a la realización de un movimiento sobre el $path[i]$ determinado mediante $compute\_constraintpath$ se realiza una visualización por cada agente sobre su entorno.

El proceso $obseve\_agent2$ junto a $observe\_new\_agents$ consiste en que si se pierde de vista un agente j por parte de un agente i, entonces se eliminan los registros de este, ya que no se ven entre ellos. En caso de encontrarse ambos i y j a la vista, el agente i evalúa el comportamiento del agente j en base a las decisiones planeadas por j, es decir, el agente i espera que el agente j realice su próximo movimiento sobre la celda siguiente dentro del $path[j]$, es por ello que i actualiza de manera predictiva los contadores $fromTransition$, $toTransition$, $blockedAgent$ y $agentMovingTo$ en base a que i espera

que *j* realice este movimiento sobre la siguiente ventana de tiempo. Esta predicción realizada será empleada dentro de la siguiente iteración del algoritmo para determinar los costos de conflicto.

Anexo 8

Conceptos básicos sobre la arquitectura de un computador [5]. Entre los componentes principales que forman parte de un computador son:

- *CPU*.
- Memoria *RAM*.
- Memoria de almacenamiento secundaria.

La *CPU* se encarga de ejecutar las instrucciones básicas en lenguaje binario, esto mediante las llamadas realizadas por el sistema operativo haciendo uso de un lenguaje de programación a bajo nivel.

En tanto a la *Random Access Memory (RAM)* o memoria de acceso aleatorio, en esta se cargan todas las instrucciones a ejecutarse por la CPU y otras unidades presentes en el computador, su naturaleza es volátil, por lo que al quitar el suministro de energía los datos almacenados en ella se perderán.

En última instancia se cuenta con la memoria secundaria o de almacenamiento persistente, esta permite almacenar datos de gran tamaño que se requerirán a futuro, como lo son el sistema operativo, hojas de cálculo, software, etc. Esta suele presentarse como unidades de disco duro *HDD*, unidades de almacenamiento en estado sólido *SSD* y unidades portables como Pendrives o memorias *SD*.

Por el lado del software que utiliza un computador se deben destacar dos elementos. El primero es el sistema operativo, este permite ejecutar aplicaciones que sean controladas por el usuario, para luego mediante el *kernel* (que comúnmente acompaña al sistema operativo) se realicen las llamadas a bajo nivel o a nivel ensamblador, que permitan ejecutar estas rutinas. El segundo elemento importante es el compilador, los lenguajes de programación compilados como lo son *C* y *C++* deben hacer uso de un compilador que sirva de traductor de las rutinas implementadas por el usuario a un lenguaje ensamblador de bajo nivel, que sea capaz de ser leído y ejecutado por la *CPU*.