PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

# COMPILACIÓN EN PROGRAMACIÓN DE CONJUNTOS DE RESPUESTAS PARA EL PROBLEMA DE BÚSQUEDA DE CAMINOS CON MÚLTIPLES AGENTES

**RODRIGO NICOLÁS TEÓFILO GÓMEZ ARAYA**

Tesis para optar al grado de
Magíster en Ciencias de la Ingeniería

Profesor Supervisor:
JORGE BAIER

Santiago de Chile, Diciembre 2020

© MMXX, RODRIGO NICOLÁS TEÓFILO GÓMEZ ARAYA

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

# COMPILACIÓN EN PROGRAMACIÓN DE CONJUNTOS DE RESPUESTAS PARA EL PROBLEMA DE BÚSQUEDA DE CAMINOS CON MÚLTIPLES AGENTES

**RODRIGO NICOLÁS TEÓFILO GÓMEZ ARAYA**

Miembros del Comité:
JORGE BAIER
JUAN REUTTER
CARLOS HERNÁNDEZ
LUCIANO CHIANG

Tesis para optar al grado de
Magíster en Ciencias de la Ingeniería

Santiago de Chile, Diciembre 2020

© MMXX, RODRIGO NICOLÁS TEÓFILO GÓMEZ ARAYA

*A mi mamá, papá y hermana, los quiero mucho.*

## AGRADECIMIENTOS

Me es muy difícil escribir las palabras adecuadas para describir lo mucho que estoy agradecido con mi profesor guía Jorge. Desde la primera clase con él hasta el día de hoy ha sido una parte fundamental en mi desarrollo académico, gracias al apoyo de él esta investigación no habría podido ser posible. También, por sobre todas las cosas, quiero agradecerle por ser ser el principal pilar de mi bienestar emocional, gracias a su comprensión, compañía y preocupación constante es que puedo escribir estas palabras hoy.

Agradezco también a quienes de alguna manera u otra han sido parte de investigación. Principalmente, gracias a Carlos por las múltiples discusiones y consejos que permitieron hacer posible nuestras publicaciones.

También quiero agradecer el apoyo de todos mis amigos y compañeros de magíster que estuvieron conmigo este último tiempo. Las infinitas conversaciones, consejos, y por sobre todo, compañía, me han dado la fuerza y alegría necesarias para sobrevivir estos últimos 3 años.

Muchas gracias también a mi psicóloga Begoña, por ayudar en mi desarrollo personal, por estar ahí cuando más lo necesitaba, y ser una de las guías más importantes en mi camino hacia la madurez. Gracias a ella es que después de tantos años de oscuridad puedo ver una ventana hacia la luz.

Gracias a mi familia, a mi mamá Joan por darme tanto cariño y amor durante toda mi vida, a pesar de todos los problemas sé que siempre estarás ahí esperándome con una sonrisa. A mi papá Rodrigo, por siempre preocuparse de mí y de que no me falte nada. A mi hermana Paula, por todos los años que hemos compartido y ser una de las principales fuentes de alegría en mi vida.

Por último, quiero agradecerme a mí mismo, por haber aguantado tantos años, por seguir adelante. Quiero dejar este texto como un recordatorio para el futuro: *No te rindas.*

## ÍNDICE DE CONTENIDOS

AGRADECIMIENTOS IV

ÍNDICE DE FIGURAS VIII

ÍNDICE DE TABLAS X

ABSTRACT XII

RESUMEN XIII

1. INTRODUCCIÓN 1

2. PRELIMINARES 7

2.1. Búsqueda en un solo agente . . . . . 7

2.1.1. BFS . . . . . 8

2.1.2. A* . . . . . 11

2.2. MAPF . . . . . 13

2.2.1. Utilizando BFS para calcular heurísticas . . . . . 15

2.3. Solvers de MAPF . . . . . 17

2.3.1. Algoritmos basados en búsqueda . . . . . 17

2.3.2. Algoritmos basados en transformaciones . . . . . 20

2.4. ASP . . . . . 21

2.5. Optimizadores . . . . . 24

2.5.1. Branch-and-Bound . . . . . 24

2.5.2. Unsatisfiable Core . . . . . 24

3. CODIFICACIÓN 25

3.1. Compilación Básica . . . . . 25

3.1.1. Átomos . . . . . 25

3.1.2. Especificación de la Instancia . . . . . 26

3.1.3. Efectos . . . . . 26

3.1.4. Ejecución de movimientos en paralelo . . . . . 27
3.1.5. Posiciones legales . . . . . 27
3.1.6. Conflictos de Vértices . . . . . 28
3.1.7. Conflictos de intercambio de agentes . . . . . 29
3.1.8. Logro del Objetivo . . . . . 29
3.1.9. Tamaño de la codificación Básica . . . . . 29
3.2. Suma de costos en ASP . . . . . 29
3.2.1. Codificación dependiente de la grilla . . . . . 30
3.2.2. Codificación independiente de la grilla . . . . . 31
3.3. Una Codificación Lineal . . . . . 32
3.4. Utilizando búsqueda para reducir el número de átomos . . . . . 34
3.5. Encontrando Soluciones Óptimas . . . . . 34

4. EXPERIMENTOS 38
4.1. Experimentos sobre distintas configuraciones . . . . . 39
4.1.1. Grillas de $N \times N$ con obstáculos aleatorios . . . . . 39
4.1.2. Experimentos en Bodegas . . . . . 42
4.2. Soluciones makespan-óptimas . . . . . 44
4.3. Grillas $N \times N$ Libres de Obstáculos . . . . . 53

5. EPANDIENDO ASP 57
5.1. Solvers Offline . . . . . 57
5.2. Solvers Online . . . . . 58
5.3. WHCA* . . . . . 58
5.4. ASP con ventanas . . . . . 60
5.4.1. Nuevos Átomos . . . . . 60
5.4.2. Codificación de los efectos . . . . . 61
5.4.3. Logro del Objetivo . . . . . 62
5.4.4. Optimización de costos . . . . . 62
5.4.5. Algoritmo . . . . . 64

6. EXPERIMENTOS CON VENTANAS 67
6.1. Mapa den520d . . . . . 68
6.2. Mapa ost003d . . . . . 75
6.3. Mapa brc202d . . . . . 80
7. CONCLUSIONES 84
BIBLIOGRAFÍA 86

## ÍNDICE DE FIGURAS


1.1. Ejemplo de un sistema de bodegas. . . . . 1

1.2. Ejemplo de estados sucesores posibles en una grilla de 4x4. . . . . 2

2.1. Ejemplo de problema de búsqueda en un solo agente. . . . . 8

2.2. Árbol de búsqueda de BFS hasta profundidad 3 en problema de ejemplo . . . 9

2.3. Problema donde el aumento de makespan genera mejores soluciones. . . . . 15

4.1. Tasa de éxito y número de instancias resueltas en grillas de $20 \times 20$ con distinto número de agentes. . . . . 41

4.2. Tasa de éxito y número de instancias resueltas en grillas de $20 \times 20$ con distinto porcentaje de obstáculos. . . . . 41

4.3. Ejemplo de problema de bodega. . . . . 43

4.4. Tasa de éxito y número de instancias resueltas en mapa de bodega. . . . . 43

4.5. Tasa de éxito y número de instancias resueltas a medida que el número de agentes varía en una grilla de $20 \times 20$. . . . . 48

4.6. Tasa de éxito y número de instancias resueltas a medida que el porcentaje de obstáculos varía en una grilla de $20 \times 20$. . . . . 48

4.7. Tasa de éxito y número de instancias resueltas a medida que el número de agentes varía en mapa de bodega. . . . . 49

4.8. Tasa de éxito versus ICBS-h en grillas libres de obstáculos de distinto tamaño. 53

4.9. Tasa de éxito en grilla de $20 \times 20$, tiempo límite fijado en 15 segundos. . . . 55

4.10. Tiempo total en escala logarítmica de ejecución e instanciación sobre grillas de distinto tamaño. . . . . 56

4.11. Tiempo total en escala logarítmica de ejecución e instanciación sobre grillas de $20 \times 20$, tiempo límite fijado en 15 segundos. . . . . 56

6.1. Mapas DAO . . . . . 68

6.2. Tasa de éxito y número de instancias resueltas en mapa den520d utilizando ASP. . . . . 71

6.3. Tasa de éxito y número de instancias resueltas en mapa den520d utilizando WHCA*. . . . . 71

6.4. Tiempo de ejecución e instanciación en mapa den520d. . . . . 74

6.5. Tasa de éxito y número de instancias resueltas en mapa ost003d utilizando ASP. 76

6.6. Tasa de éxito y número de instancias resueltas en mapa ost003d utilizando WHCA*. . . . . 76

6.7. Tiempo de ejecución e instanciación en mapa ost003d. . . . . 79

6.8. Tasa de éxito y número de instancias resueltas en mapa brc202d utilizando ASP. . . . . 81

6.9. Tasa de éxito y número de instancias resueltas en mapa brc202d utilizando WHCA*. . . . . 81

6.10. Tiempo de ejecución e instanciación en mapa brc202d. . . . . 83

## ÍNDICE DE TABLAS

4.1. Suboptimalidad de las soluciones makespan-óptimas en grillas de $20 \times 20$ variando el número de agentes. . . . . 45

4.2. Suboptimalidad de las soluciones makespan-óptimas en grillas de $20 \times 20$ variando el porcentaje de obstáculos. . . . . 46

4.3. Suboptimalidad de las soluciones makespan-óptimas en mapa de bodega. . . 46

4.4. Esfuerzo necesario para encontrar la solución costo-óptima en grilla $20 \times 20$ con un número creciente de agentes. . . . . 51

4.5. Esfuerzo necesario para encontrar la solución costo-óptima en grilla $20 \times 20$ con un porcentaje creciente de obstáculos. . . . . 52

4.6. Esfuerzo necesario para encontrar la solución costo-óptima en mapa de bodega con un número creciente de agentes. . . . . 52

6.1. Costo promedio sobre mapa den520d. . . . . 72

6.2. Tiempo de solución promedio sobre mapa den520d. . . . . 72

6.3. Suboptimalidad promedio sobre mapa den520d. . . . . 73

6.4. Tiempo de instanciación promedio en mapa den520d. . . . . 73

6.5. Costo promedio sobre mapa ost003d. . . . . 77

6.6. Tiempo de solución promedio sobre mapa ost003d. . . . . 77

6.7. Suboptimalidad promedio sobre mapa ost003d. . . . . 78

6.8. Tiempo de instanciación promedio en mapa ost003d. . . . . 78

6.9. Costo promedio sobre mapa brc202d. . . . . 82

6.10. Tiempo de solución promedio sobre mapa brc202d. . . . . 82

6.11. Tiempo de instanciación promedio en mapa brc202d. . . . . 83

## ABSTRACT

Multi-agent pathfinding (MAPF) is the problem of finding $k$ non-colliding paths connecting $k$ given initial positions with $k$ given goal positions on a given map. In its sum-of-costs variant, the total number actions performed by agents before they definitely reach the goal is minimized. Not surprisingly, since MAPF is combinatorial, a number of compilations to Boolean Satisfiability (SAT) and Answer Set Programming (ASP) exist. In this thesis, we describe in detail the first family of compilations to ASP that solve sum-of-costs MAPF over 4-connected grids. Compared to existing ASP compilations, a distinguishing feature of our compilation is that the number of total clauses (after grounding) grow *linearly* with the number of agents, while existing compilations grow quadratically. In addition, the optimization objective is such that its size after grounding does not depend on the size of the grid. In our experimental evaluation, we show that our approach outperforms the state of the art solvers when grids are congested with agents. Finally, we show an online variant of our compilation to ASP that allows to solve problems of greater size and number of agents.

**Keywords:** Multi-Agent Pathfinding, Answer Set Programming, Heuristic Search, Planning, Optimization, Logic Programming, Combinatorial Problems.

# RESUMEN

La búsqueda de caminos con múltiples agentes (MAPF por sus siglas en inglés) es el problema de encontrar $k$ caminos libres de conflictos que conecten $k$ posiciones iniciales con $k$ posiciones objetivos en un mapa dado. En su variante de suma de costos, se minimiza el número total de acciones realizadas por los agentes. Dado que MAPF es un problema combinatorio, existan distintas compilaciones a Satisfacción Booleana (SAT) y Programación de Conjuntos de Respuestas (ASP). En esta tesis, describimos en detalle la primera familia de compilaciones a ASP que resuelven la variante de suma de costos de MAPF sobre grillas 4-conectadas. En comparación con otras compilaciones existentes de ASP, nuestra compilación se diferencia en que el número de cláusulas totales (después de la instanciación) crece *linealmente* con el número de agentes, mientras que las compilaciones existentes crecen de forma cuadrática. Además, el objetivo de optimización es tal que su tamaño después de la instanciación no depende del tamaño de la grilla. Nuestra evaluación experimental muestra que nuestro enfoque supera al estado del arte cuando las grillas están congestionadas con agentes. Finalmente, mostramos una variante online de nuestra compilación que permite solucionar problemas de mayor tamaño y número de agentes.

**Palabras Claves:** Búsqueda de Caminos con Múltiples Agentes, Programación de Conjuntos de Respuestas, Búsqueda Heurística, Planeamiento, Optimización, Programación en Lógica, Problemas Combinatorios.

# 1. INTRODUCCIÓN

El trabajo de esta tesis nace de la motivación por resolver problemas como los planteados por Kiva Systems, una compañía fundada en 2003 [1] con el objetivo de desarrollar sistemas robóticos que automaticen la obtención y entrega de artículos en bodegas. La Figura 1.1 ilustra un ejemplo clásico de una bodega de Amazon, las cuales tienen estaciones de entrega con inventarios y lugares de almacenamientos donde robots se mueven de forma autónoma para llevar los artículos almacenados en las bodegas hacia las estaciones y así lograr cumplir con las entregas.

Aquí nace la necesidad de elaborar algoritmos permitan automatizar el trabajo de estos robots, los cuales pueden ser modelados según un *problema de búsqueda de caminos multi agente* (MAPF por sus siglas en inglés *Multi-Agent Pathfinding*). Este modelo ha sido ampliamente investigado en el área de la Inteligencia Artificial y tiene distintas aplicaciones en la vida real, desde la automatización de cientos de robots moviéndose en conjunto dentro una bodega, hasta aviación, minería y videojuegos (Wang y Botea, 2008).

Existen distintas versiones de MAPF, por ejemplo, se pueden modelar las estaciones para que tengan una lista de inventarios, almacenes con un stock limitado de artículos, agentes con distintos tamaños y funciones de costos, etc. En esta tesis se trabajará sobre

[1] Luego adquirida por Amazon en 2012

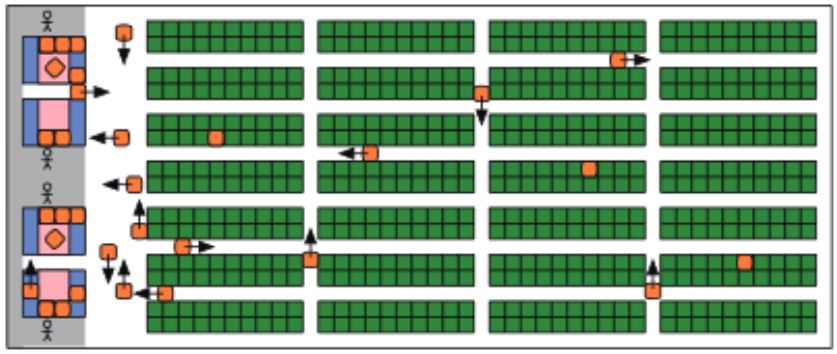


Figura 1.1. Ejemplo de un sistema de bodegas.

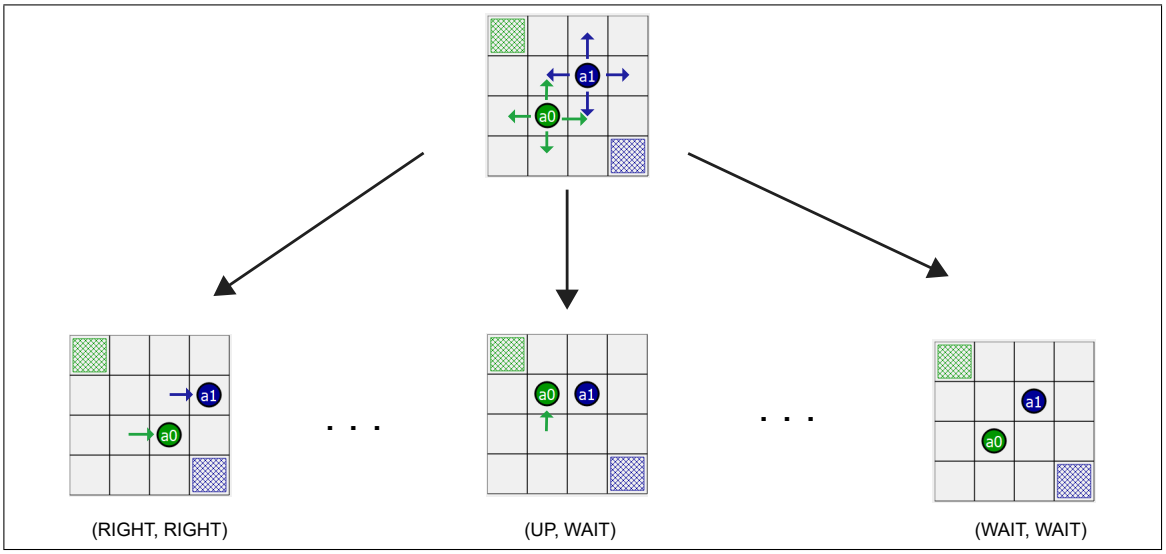



Figura 1.2. Ejemplo de estados sucesores posibles en una grilla de 4x4.

la versión más básica de MAPF: dada una grilla que representa una bodega y un número de agentes, donde cada uno de estos está asociado a una celda inicial y una celda objetivo en la grilla. El problema consiste en encontrar todos los caminos libres de conflictos que conecten a la celda inicial y objetivo para cada uno de los agentes, minimizando el costo total de estos caminos.

Consideremos por ejemplo una instancia inicial con $2$ agentes en una grilla de $4 \times 4$. Por cada instante de tiempo los agentes se pueden mover en las 4 direcciones cardinales o quedarse esperando en sus celdas, teniendo en cuenta esto, en el siguiente instante de tiempo son generados $5^2 = 25$ estados sucesores posibles. Estos se generan según todas las combinaciones de acciones que pueden tomar los agentes, por ejemplo, como muestra la Figura 1.2, ambos agentes se pueden mover hacia la derecha, o uno se puede quedar esperando en su celda, o ambos se pueden quedar esperando en su celda, o cualquiera de todas las posibles $25$ combinaciones de acciones.

Por otro lado, es fácil ver que la cantidad de configuraciones posibles sobre las posiciones de los agentes son $16 \times 15 = 240$, ya que el primer agente puede posicionarse en

cualquier celda y el siguiente agente se puede posicionar en cualquier otra celda distinta a la del primer agente.

Ahora, si se agregara un agente extra a la grilla, entonces la cantidad de estados sucesores aumentaría a $5^3 = 125$ y la cantidad de configuraciones posibles a $16 \times 15 \times 14 = 3360$.

Entonces, no es sorprendente el hecho de que resolver MAPF de manera óptima sea NP-hard (Yu y LaValle, 2013; Ma y Koenig, 2017) y que la construcción de algoritmos que escalen relativamente bien según el número de agentes sea tan desafiante.

Para esta versión de MAPF se han propuesto distintos enfoques, pero dos clases de *solvers* son los más relevantes para la investigación que hemos realizado. La primera clase son los solvers basados en búsqueda (p. ej. (Standley, 2010)), los cuales utilizan búsqueda heurística como su componente principal. Uno de los algoritmos que está en el estado del arte es *Conflict-Based Search* (CBS) (Sharon, Stern, Felner, y Sturtevant, 2012; Felner y cols., 2018; Li, Felner, Boyarski, Ma, y Koenig, 2019). Normalmente estos solvers resuelven MAPF minimizando la suma del largo de los caminos entre todos los agentes, en otras palabras, buscan optimizar el rendimiento de los agentes, haciendo que ejecuten la menor cantidad de acciones totales. Este valor está definido como la *suma de costos* de una solución.

La segunda clase son los solvers basados en compilaciones, que por ejemplo transforman MAPF a un problema de Satisfacción de fórmulas booleanas (SAT). (entre ellos, (Surynek, Felner, Stern, y Boyarski, 2016); (Barták, Zhou, Stern, Boyarski, y Surynek, 2017); (Barták y Svancara, 2019)), y *Answer-Set Programming* (ASP) (Erdem, Kisa, Öztok, y Schüller, 2013; Gebser y cols., 2018). La gran mayoría de estos resuelven MAPF minimizando el largo del camino más largo entre todos los agentes, es decir, buscan optimizar el tiempo total necesario para que todos los agentes lleguen a sus respectivos objetivos. Este valor está definido como el *makespan* de una solución.

También los solvers para problemas de búsqueda pueden ser clasificados según como realizan sus fases de planeamiento y movimiento. Decimos que un solver es *offline* si es

que realiza las fases de planeamiento y movimiento de forma totalmente independiente y solo una vez. Por otro lado, un solver *online* intercala estas fases constantemente, realizando primero un planeamiento parcial para luego ejecutar algunos movimientos, repitiendo este proceso hasta llegar a una solución completa.

Los solvers online surgen debido a que las instancias de MAPF pueden ser muy complicadas para ser resueltas por los solvers clásicos offline. Los solvers online permiten resolver estas instancias, reduciendo considerablemente el tiempo de resolución, pero, sacrificando la optimalidad de las soluciones.

En la primera parte de esta tesis nos enfocamos en resolver los problemas de manera offline. Continuamos explorando el potencial de los solvers en ASP para MAPF, y proponemos la primera compilación en ASP que resuelve MAPF optimizando el largo de los caminos, bajo la función de suma de costos, además entregan la ventaja de ser la primera compilación en ASP que crece linealmente en el número de agentes, a diferencia de las compilaciones existentes que son cuadráticas en el número de agentes. Además, proponemos una optimización que utiliza información obtenida desde un algoritmo de búsqueda en forma de precómputo para hacer la compilación aún más compacta.

En nuestra evaluación empírica, probamos nuestro enfoque en grillas cuadradas y bodegas con un número creciente de agentes. Nos comparamos con MDD-SAT (Surynek y cols., 2016), un solver representativo del estado del arte en algoritmos basados en compilaciones, e ICBS-h (Felner y cols., 2018), que es representativo del estado del arte de MAPF basado en búsqueda. Observamos que nuestro enfoque supera tanto a MDD-SAT e ICBS-h cuando hay alta congestión. Específicamente, nuestro enfoque tiene una mejor cobertura (proporción de problemas resueltos) a medida que el número de agentes aumenta. Adicionalmente, analizamos el rendimiento de nuestro solver cuando encontramos las llamadas soluciones *makespan-óptimas*, es decir, soluciones en las que primero se minimiza el *makespan* y luego se minimiza el costo.

En la segunda parte, mostramos una versión online del solver propuesto anteriormente, que, utilizando ideas de algoritmos con ventanas, logra solucionar y escalar sobre problemas en grillas de tamaño y número de agentes mayor. Observamos que nuestro solver no logra competir en términos de rendimiento con el estado del arte, sin embargo, encuentra soluciones con costos mucho más cercanos a los óptimos.

Concluimos que Answer-Set Programming es un enfoque viable para resolver problemas MAPF, especialmente en problemas con alta congestión. Otra conclusión interesante es que ASP proporciona una representación muy compacta y elegante de MAPF. De hecho, todo el código necesario para resolver una instancia de MAPF está incluido en esta tesis. Concluimos además que encontrar soluciones makespan-óptimas permite a los solvers basados en ASP encontrar soluciones más rápidamente y escalar mejor, sacrificando menos del $1\,\%$ de la calidad media de la solución. Esto es importante ya que sugiere que centrarse en soluciones óptimas en cuanto a costos en lugar de soluciones óptimas en cuanto a makespan, podría no ser un enfoque práctico.

Finalmente, las principales contribuciones de este trabajo las podemos resumir en:

- Generamos la primera compilación de ASP para MAPF que optimiza bajo la función de suma de costos.
- Independiente de la función de optimización, proponemos la primera codificación de tamaño lineal sobre el número de agentes en ASP. Además de otras optimizaciones, como una codificación inteligente de objetivos, y el uso de algoritmos de búsqueda en una fase de precómputo.
- Proponemos un solver que supera al estado del arte en problemas pequeños con alta congestión. Lo cual en la práctica podría traducirse en mejoras a solvers de versiones más complejas y cercanas a la realidad de estos problemas.
- Analizamos el potencial de las soluciones *makespan-óptimas*, que entregan soluciones muy cercanas al óptimo de la función de suma de costos.

- Proponemos una versión online de nuestro solver, el cual permite resolver problemas sobre grillas de tamaños más grandes que no logran ser resueltos por su contra parte offline.

# 2. PRELIMINARES

## 2.1. Búsqueda en un solo agente

Un problema de búsqueda de caminos tradicional con un solo agente consiste en:

- Un grafo $G = (V, E)$.
- Una función de costo por cada vértice $c : E \to \mathbb{R}^+$.
- Un vértice inicial $v_0$.
- Un conjunto de vértices objetivo $goals \subseteq V$.

El objetivo de este problema es encontrar un camino sobre $G$, de tal forma se llegue a uno de los vértices objetivo. Formalmente, un camino es una secuencia de vértices en $V$, $v_1 v_2 \ldots v_n$, donde $(v_i, v_{i+1}) \in E$ para cada $i \in \{1, \ldots, n-1\}$. Un camino $\pi = v_1 v_2 \ldots v_n$ es una solución del problema si es que $v_n \in goals$.

El costo de un camino está definido como $cost(\pi) = \sum_{i \in n} c(v_{i-1}, v_i)$. Decimos que un camino es óptimo si es que no existe otro camino $\rho$ tal que $cost(\rho) < cost(\pi)$.

En esta tesis nos centraremos en problemas donde $G$ es una grilla que solo permite movimientos en 4 direcciones posibles (*4-conectada*): $izquierda$, $derecha$, $arriba$, $abajo$. También asumiremos que la función de costos es uniforme, es decir, donde todas las acciones tienen el mismo valor (sin pérdida de generalidad, podemos asumir que $c(e) = 1$ por cada $e \in E$). En la Figura 2.1 se ilustra un ejemplo, en donde se debe ir desde la posición $(0, 2)$ hasta la posición $(4, 2)$

Existen diversos algoritmos para resolver el problema de búsqueda en un solo agente. A continuación, mostraremos dos relevantes al trabajo en esta tesis.

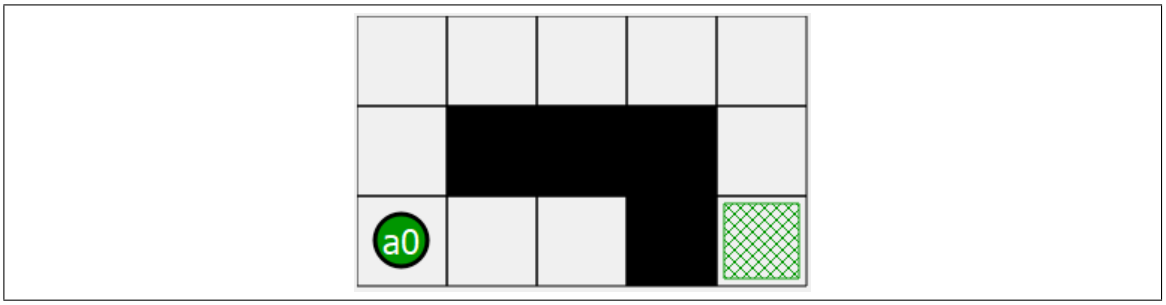



Figura 2.1. Ejemplo de problema de búsqueda en un solo agente.

### 2.1.1. BFS

Uno de los algoritmos clásicos para resolver búsqueda con costos uniformes es BFS (*Breadth-First Search* o búsqueda en amplitud). Este algoritmo es completo y garantiza encontrar caminos óptimos en términos de costo.

Este algoritmo consiste en recorrer el grafo en niveles que tienen igual valor. Donde primero se recorren los vértices que están a distancia 1, luego los que están a distancia 2, y así sucesivamente hasta llegar a un vértice objetivo. Tomando el ejemplo de la sección anterior, el árbol de búsqueda se ve como muestra la Figura 2.2.

A cada vértice $v$, se le asigna un costo $v.cost$ que indica el costo que hay desde el vértice inicial hasta $v$, y un padre $v.parent$ que indica el vértice antecesor en el camino óptimo. En un inicio, se le asigna a cada vértice un costo $v.cost = \infty$ y un padre $v.parent = null$, exceptuando el vértice inicial que tiene costo $0$.

Luego, el vértice inicial es agregado a una cola llamada $OPEN$, la cual representa la frontera de la búsqueda, donde a medida que va avanzando el algoritmo se van agregando los vértices que han sido encontrados a esta cola.

El ciclo del algoritmo parte eligiendo y removiendo el primer vértice $v$ de $OPEN$ (acción llamada expansión de vértice). Después se revisa que $v$ sea un vértice objetivo. Si no lo es, agrega al final de $OPEN$ todos los vecinos de $v$ que todavía no han sido agregados ni expandidos (acción llamada generación de vértice), sumando 1 al costo del

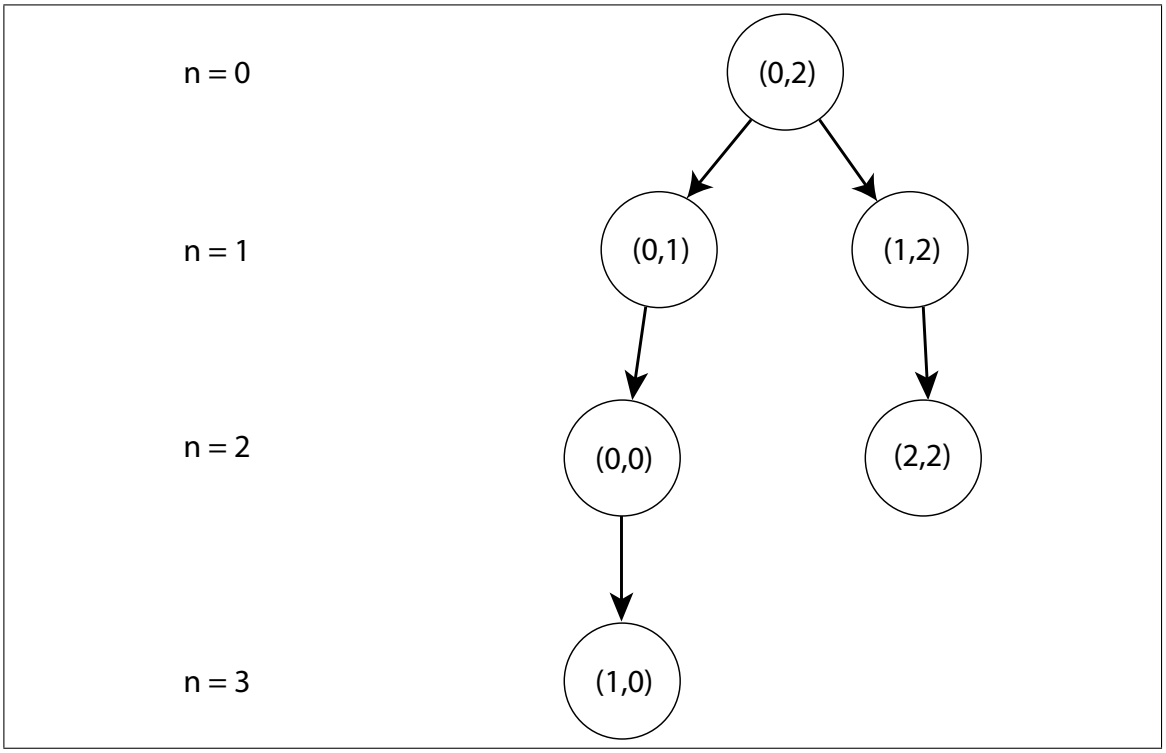



Figura 2.2. Árbol de búsqueda de BFS hasta profundidad 3 en problema de ejemplo

padre y asignando como padre a $v$. En caso de que $v$ sea objetivo, entonces el algoritmo termina y retorna el vértice (donde el camino óptimo puede ser encontrado recorriendo los vértices padres de $v$). El pseudo código esta descrito en Algoritmo 1

Es importante mencionar que BFS además de entregar un camino óptimo hasta un vértice objetivo también dejará almacenado en $v.cost$ y $v.parent$ la información del camino óptimo entre $v_0$ y $v$. Es más, si se indica que $goals = \emptyset$, entonces BFS almacenará todas las distancias y caminos óptimos entre todos los vértices alcanzables desde $v_0$.

**Algoritmo 1**: BFS

```
Input: (G, v_0, goals)
Output: Un camino desde v_0 hasta un v ∈ goals
forall v ∈ G do
    v.cost ← ∞
    v.parent ← null
end
OPEN ← Una cola vacía
v_0.cost ← 0
OPEN.encolar(v_0)
while OPEN ≠ ∅ do
    v ← OPEN.desencolar()
    if v ∈ goals then
        return v
    end
    forall w que está conectado desde v do
        if w.cost > v.cost + 1 then
            w.cost ← v.cost + 1
            w.parent ← v
            OPEN.encolar(w)
        end
    end
end
return null
```

### 2.1.2. A*

Cuando la función de costos no es uniforme uno de los algoritmos más populares para resolver este problema es A* (Hart, Nilsson, y Raphael, 1968). De forma similar a BFS utiliza una cola $OPEN$ para elegir el orden de expansión. Sin embargo, este explora el grafo de manera *best-first*, es decir, funciona expandiendo los vértices más prometedores dada cierta función.

En cada iteración del algoritmo A* determina que vértice expandir seleccionando el camino que minimiza la función $f(v) = g(v) + h(v)$, donde $v$ es un vértice del grafo, $g(v)$ es el costo desde el vértice inicial hasta $v$ y $h(v)$ es una función heurística que estima el costo mínimo desde $v$ hasta el objetivo más cercano.

Si $h$ es una heurística admisible, es decir que no sobre estima los costos óptimos ($h(v) < c^*(v)$, donde $c^*(v)$ es el costo óptimo desde $v$ hasta un objetivo). Entonces está garantizado que A* entrega una solución de costo óptima. El pseudo código esta descrito en Algoritmo 2

Cabe destacar que cuando se tiene una heurística informativa (es decir, cuando $h(v)$ tiene valores cercanos a $c^*(v)$), usualmente es más eficiente usar A* que BFS para encontrar caminos óptimos con costos uniformes.

**Algoritmo 2**: A*

**Input**: $(G, v_0, goals, c, h)$
**Output**: Un camino desde $v_0$ hasta un $v \in goals$

```
forall v ∈ G do
    v.g ← ∞
    v.f ← ∞
    v.parent ← null
end
OPEN ← Una cola priorizada vacía basada en f
v_0.g ← 0
v_0.f ← h(v_0)
OPEN.encolar(v_0)
while OPEN ≠ ∅ do
    v ← OPEN.desencolar() según el menor valor de f
    if v ∈ goals then
        return v
    end
    forall w que está conectado desde v do
        g' ← v.g + c(v, w)
        if w.cost > g' then
            w.g ← g'
            w.f ← g' + h(w)
            OPEN.encolar(w)
            Reordernar OPEN con el nuevo valor de w.f
        end
    end
end
return null
```

## 2.2. MAPF

MAPF o en sus siglas en inglés *Multi Agent Path-Finding* (Búsqueda de caminos con múltiples agentes), es equivalente al problema descrito anteriormente, pero ahora el objetivo es encontrar caminos para varios agentes en un mismo espacio sin que ellos tengan conflictos.

Formalmente, una instancia de MAPF está definida por una tupla $(G, \mathcal{A}, init, goal)$, donde $G = (V, E)$ es un grafo, $A$ es el conjunto de agentes, e $init : A \to V$ y $goal : A \to V$ son funciones para denotar los vértices de inicio y objetivo por cada uno de los agentes.

En cada instante de tiempo, cada agente que se encuentra en un vértice $v$ puede moverse a cualquiera de sus sucesores en $G$ o no moverse (a diferencia de la búsqueda en un solo agente). Cuando $G$ es una grilla 4-conectada hay 5 posibles acciones en cada instante de tiempo: $\{arriba, abajo, izquierda, derecha, esperar\}$, donde las primeras 4 mueven al agente en una de las 4 direcciones cardinales, y la acción $esperar$ hace que el agente se mantenga en el vértice actual.

Un camino sobre $G$ es una secuencia de vértices en $V$, $v_1 v_2 \ldots v_n$, donde $v_i = v_{i+1}$ (es decir, se ejecutó una acción $esperar$) o $(v_i, v_{i+1}) \in E$ (es decir, una acción distinta a $esperar$ fue realizada) para cada $i \in \{1, \ldots, n-1\}$. Dado un camino $\pi$, denotamos a $\pi[i]$ como $i$-ésimo elemento en $\pi$, donde $i \in \{1, \ldots, |\pi|\}$.

Una solución a una instancia de MAPF es una función $sol : \mathcal{A} \to V^*$, la cual asocia un camino por cada uno de los agentes, tal que el primero y último vértice de $sol(a)$ son, respectivamente, $init(a)$ y $goal(a)$. Sin pérdida de generalidad, desde ahora en adelante asumiremos que todos los caminos en $sol$ tienen el mismo tamaño, ya que los agentes pueden realizar $esperar$ al final de toda secuencia de acciones para permanecer en el mismo vértice. Denotamos $\mathcal{T}$ como el conjunto $\{1, \ldots, M\}$, donde $M$ es el tamaño de los caminos en $sol$. Así también nos referiremos a $M - 1$ como el *makespan* de la solución.

Adicionalmente, $sol$ debe estar *libre de conflictos*, esto quiere decir que dados $\pi, \rho \in sol$ que representan los caminos de dos agentes distintos, no puede ocurrir ninguno de los siguientes conflictos:

- **Conflicto de vértices**. Dos agentes no pueden estar en el mismo vértice en el mismo instante de tiempo. Formalmente, existe un conflicto de este tipo si y solo si $\pi[i] = \rho[i]$, para algún $i \in \mathcal{T}$.
- **Conflicto de Intercambio**. Dos agentes no pueden intercambiar de posiciones. Intuitivamente, este conflicto se justifica por el hecho de que el tamaño de los agentes previene la conexión entre dos vértices en direcciones opuestas. Formalmente, existe un conflicto de este tipo, si y solo si $(\pi[i], \rho[i]) = (\rho[i+1], \pi[i+1])$, para algún $i \in \{1, \ldots, M-1\}$.

Tomando como base la mayoría de la literatura existente sobre MAPF (por ejemplo, Sharon y cols. (2012)), solo consideramos estos dos tipos de conflictos, y ninguno más. Por ejemplo, no consideramos los llamados *Conflictos de seguimiento* (Stern y cols., 2019), que no permiten a los agentes ocupar un vértice en el instante $t+1$ si es que otro agente estuvo en ese vértice en el instante $t$.

Un supuesto estándar en MAPF es que todos los movimientos tienen costo 1 con excepción de las acciones $esperar$ hechas en el objetivo cuando ninguna otra acción es planificada a futuro. Así, el costo de un camino $\pi$ por cada agente $a$ se escribe como $|\rho|$, donde $\rho$ es la secuencia más pequeña tal que $\pi = \rho(goal(a))^k$ para algún $k$. El costo de la solución $sol$ está definida como $\sum_{a \in A} c(sol(a))$.

Una solución $sol$ es óptima bajo *suma de costos*, o simplemente *costo-óptima*, si es que no existe otra solución $sol'$ tal que $c(sol') < c(sol)$. Por otra parte, una solución $sol$ es *makespan-óptima*, si es que no existe otra solución tal que su makespan es más pequeño que el makespan de $sol$. Es importante notar que una solución makespan-óptima no garantiza necesariamente una solución costo-óptima. Esto se ve ilustrado en la Figura 2.3, donde el problema tiene 3 agentes: $a_0$ que necesita ir desde $(0,1)$ a $(3,1)$, y los agentes $a_1$ y $a_2$,

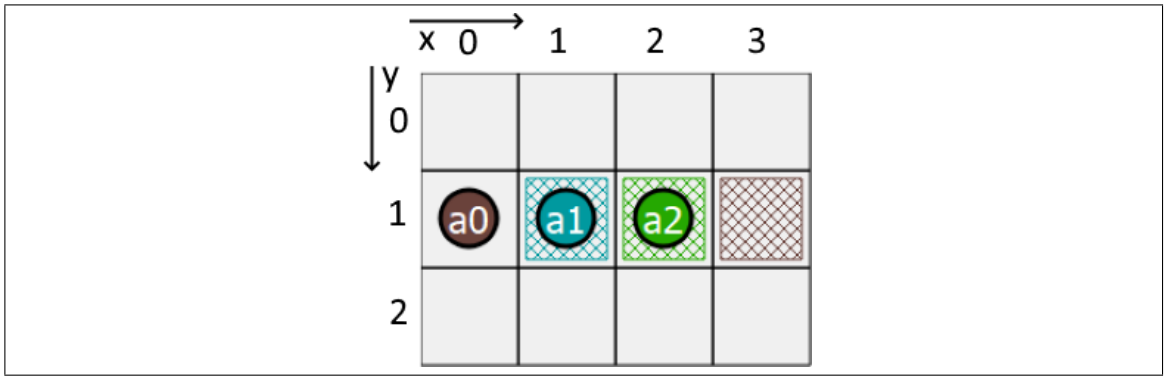



Figura 2.3. Problema donde el aumento de makespan genera mejores soluciones.

que ya se encuentran sus celdas objetivo en las posiciones $(1, 1)$ y $(2, 1)$ respectivamente. La solución de costo-óptima con makespan $3$ es $8$, dado que requiere que los agentes $a_1$ y $a_2$ se muevan fuera de su objetivo para así darle espacio a $a_0$ para que se mueva hacia su objetivo. Por otro lado, la solución con makespan $5$ es $5$, ya que solo necesita que $a_0$ se mueva.

### 2.2.1. Utilizando BFS para calcular heurísticas

Es común que los solvers de MAPF precomputen información para agilizar la resolución. Una técnica especialmente relevante para el trabajo de esta tesis consiste en separar y resolver el problema en búsquedas individuales de un solo agente, con el fin de utilizar los resultados como heurística para la resolución del problema general.

Este procedimiento se puede apreciar en el Algoritmo 3. Donde dado un problema en MAPF $(G = (V, A), \mathcal{A}, init, goal)$, el algoritmo entregará como respuesta una tabla $h$ de tamaño $|\mathcal{A}| \times |V|$, en la cual $h(a, v)$ indica la distancia óptima entre $v$ y $goal(a)$, ignorando los posibles conflictos con los demás agentes $a' \in \mathcal{A} \setminus a$.

Por cada uno de los agentes $a \in \mathcal{A}$ se realiza una búsqueda BFS sobre el grafo original, partiendo desde $goal(a)$ y hasta que todos los vértices alcanzables desde $goal(A)$ hayan sido expandidos.

**Algoritmo 3**: Precómputo de Heurísticas con BFS

**Input**: Un problema MAPF $(G = (V, A), \mathcal{A}, init, goal)$
**Output**: Tabla $h$ que contiene las distancias entre todos los vértices y objetivos

```
h ← Tabla 2D vacia de tamaño |A| × |V|
forall a ∈ A do
    G' ← G
    BFS(G', goal(a), ∅)
    forall v ∈ G'.V do
        h(a, v) ← v.cost
    end
end
return h
```

## 2.3. Solvers de MAPF

En esta sección describiremos el estado del arte en cuanto a los distintos solvers utilizados para resolver MAPF óptimamente. Describiremos estos dependiendo del método o paradigmas utilizando.

### 2.3.1. Algoritmos basados en búsqueda

Normalmente estos algoritmos están basados en el uso de variantes de A*.

#### 2.3.1.1. Utilizando A* estándar

Consiste en reducir MAPF a un problema de búsqueda de un solo agente para luego resolverlo con A*, esto se hace tratando al conjunto de agentes $\mathcal{A}$ como una sola entidad. Formalmente, dada una instancia de MAPF $(G = (V, E), \mathcal{A}, init, goal)$ creamos una instancia de búsqueda de caminos en un agente $(G' = (V', E'), c', v'_0,, goals = g')$ donde:

- $v' = (v_0, v_1, \ldots v_{n-1}) \in V' \leftrightarrow v_0, v_1, \ldots, v_{n-1} \in V$. Donde $n = |\mathcal{A}|$. Es decir, todas las combinaciones de los vértices por cada agente en la instancia original.
- $(u' = (u_0, \ldots, u_{n-1}), v' = (v_0, \ldots, v_{n-1})) \in E' \leftrightarrow (u_0, v_0), \ldots, (u_{n-1}, v_{n-1}) \in E$. Es decir, dos vértices están conectados si y solo si cada uno de los vértices originales que los componen están conectados.
- $c'(u' = (u_0, \ldots, u_{n-1}), v' = (v_0, \ldots, v_{n-1}))) = \sum_{i=0}^{n-1} c_i(u_i, v_i)$. Donde $c_i$ está definido según lo descrito en la sección anterior: $c_i(u_i, v_i) = 1$ si es que para llegar de $u$ a $v$ se requiere una acción distinta a $esperar$ y $v_i \neq goal(o)$. [1]
- $v'_0 = (init(0), \ldots init(n-1))$.
- $g = (goal(0), \ldots goal(n-1))$.

[1] También debemos agregar un costo extra si es que ejecutó $esperar$ en su objetivo pero que en un instante posterior se mueva fuera de este.

#### 2.3.1.2. Independence Detection (ID)

Debido a lo complejo en términos combinatoriales que se vuelven los problemas al tratarlos de la forma anterior, es que nace esta técnica. Esta es utilizada normalmente junto a un algoritmo de búsqueda completa como A*, permitiendo separar el problema en particiones independientes que pueden ser resueltas separadamente.

Consiste en asignar cada uno de los agentes en $\mathcal{A}$ a un conjunto $\mathcal{C}_i$. Donde $\mathcal{C}_i \cap \mathcal{C}_j = \emptyset$ para todo $i \neq j$, y $\bigcup \mathcal{G}_i = \mathcal{A}$. Es decir, cada uno de estos conjuntos son independientes y en conjunto conforman todos los agentes.

Estos conjuntos son formados de tal forma que los caminos óptimos encontrados para los agentes en un conjunto no tienen conflictos con ninguno de los otros caminos de otros conjuntos. De esta manera, utilizando A* para resolver los problemas por cada conjunto de manera independiente, se construye una solución para el problema original.

#### 2.3.1.3. Operator Decomposition (OD)

Es otra técnica que permite reducir la complejidad del primer enfoque. Este modifica el espacio de estados de forma que cada instante de tiempo es dividido por cada agente siendo necesaria la ejecución de $|\mathcal{A}|$ operadores para avanzar al siguiente instante de tiempo.

Al inicio del algoritmo de A* los agentes son ordenados de forma arbitraria. Luego, cuando un estado regular (Los estados que son generados desde la representación mostrada en la sección 2.3.1.1) es expandido, OD solo considera y aplica el movimiento al primero de los agentes de ese orden, generando un nuevo nodo intermedio. Cuando un estado intermedio es expandido, solo los movimientos de un agente son considerados, generando nuevos estados intermedios. Finalmente, cuando se expande un estado intermedio asignado al último agente, se genera un estado regular, avanzando al siguiente instante de tiempo. Esto permite disminuir el factor de ramificación de la búsqueda.

#### 2.3.1.4. CBS

CBS es la base de múltiples algoritmos (iCBS-h) en el estado del arte actual para resolver MAPF. Este consiste en separar el algoritmo en una búsqueda de dos niveles:

En el bajo nivel se realizan búsquedas de un solo agente en el mapa, ignorando los posibles conflictos con los otros agentes. Sin embargo, estas reciben como input un conjunto de restricciones que debe cumplir el camino a encontrar. Para ayudar estas búsquedas en un solo agente, como fase de preproceso CBS utiliza la técnica mostrada en la sección 2.2.1 para calcular las heurísticas.

En el alto nivel se realiza una búsqueda en un árbol binario llamado *constraint tree* (CT). En cada uno de los estados de este árbol se almacena un conjunto $C$ de restricciones. Una restricción puede ser una restricción de vértice $(a, v, t)$ que indica que el agente $a$ no puede estar en el vértice $v$ en el instante $t$, o puede ser una restricción de intercambio $(a, u, v, t)$ que prohíbe al agente $a$ moverse de $u$ a $v$ entre los instantes $t$ y $t + 1$.

Además de lo anterior, en estos nodos se almacena un conjunto de $\Pi$ de $n = |\mathcal{A}|$ caminos óptimos que satisfacen las restricciones almacenadas en este.

En un inicio, el estado raíz de este árbol contiene un conjunto vació de restricciones, y hace un llamado a la búsqueda de bajo nivel para cada uno de los agentes, entregándoles como input también un conjunto vacío de restricciones, cada uno de estos caminos es luego almacenado en $\Pi$

Luego, cuando CBS expande un estado del CT verifica que sus caminos no tengan conflictos. Donde un conflicto está representado como una tupla $(a_i, a_j, v, t)$ si es que es un conflicto de vértices, que indica que tanto como el agente $a_i$ y el agente $a_j$ ocuparon el vértice $v$ en el instante $t$. O una tupla tupla $(a_i, a_j, u, v, t)$ si es que es un conflicto de intercambio, que indica que el agente $a_i$ y el agente $a_j$ se cruzaron entre los vértices $u$ y $v$ en el instante $t$ y $t + 1$.

Si en el estado expandido no existen conflictos entonces CBS termina y entrega como solución los caminos almacenados. En caso contrario, se elige arbitrariamente un conflicto y se generan dos estados hijos:

El primero, toma al conjunto $C$ de restricciones del padre y agrega la restricción $(a_i, v, t)$ ($(a_i, u, v, t$ respectivamente), y hace un llamado a la búsqueda de bajo nivel para el agente $a_i$, entregando como input todas las restricciones que mencionan a $a_i$ en $C$ junto con la nueva restricción. Luego se modifica el conjunto $\Pi$ con el nuevo camino para ese agente.

En el caso del segundo, se repite el mismo proceso, pero utilizando al agente $a_j$.

Las variantes con mejor rendimiento de este algoritmo normalmente optimizan la búsqueda en el alto nivel a partir de heurísticas sobre la selección de conflictos o la elección de los estados sucesores.

### 2.3.2. Algoritmos basados en transformaciones

Estos algoritmos reducen en tiempo polinomial MAPF a otros problemas conocidos como SAT, Inductive Logic Programming (Yu y La Valle 2013a), ASP (Erdem et al.2013), entre otros. Normalmente estos algoritmos optimizan bajo la función de makespan, pero existen algunas variantes que funcionan bajo la función de suma de costos.

#### 2.3.2.1. MDD-SAT

Este algoritmo reduce en tiempo polinomial MAPF a un problema en SAT. Este utiliza un *Time Expansion Graph* (TEG) usualmente usado en las codificaciones a SAT que optimizan según *makespan*. En este algoritmo se propone el uso de *Multi-value Decision Diagrams* (MDD) que permiten reducir el tamaño de la codificación considerablemente, aquí se eliminan partes del TEG que no son alcanzables.

#### 2.3.2.2. Asprilo

Este entrega una codificación ASP básica para distintas variantes de MAPF. Sin embargo, no realiza optimización bajo la función de suma de costos. Además, sacrifica eficiencia por obtener una versión más general de MAPF.

## 2.4. ASP

ASP (Lifschitz, 2008) es un framework basado en lógica para resolución de problemas de optimización. A continuación, describiremos un subconjunto del estándar ASP relevante a esta tesis.

Un programa básico en ASP es un conjunto de reglas de la forma:

$$p \leftarrow q_1, q_2, \ldots, q_n. \tag{2.1}$$

Donde $n \geq 0$, y $p$, $q_1, \ldots, q_n$ son llamados *átomos*. Intuitivamente esta representación se ve como: "$p$ es verdadero/demostrable si es que $q_1, q_2, \ldots, q_n$ lo son". Cuando $n = 0$ se considera que la regla (2.1) tiene un cuerpo vacío. Estas reglas son llamadas *hechos* y usualmente son escritas como '$p$' en vez de '$p \leftarrow$'.

Un *modelo* de un programa básico de ASP es un conjunto de átomos $M$ que intuitivamente contiene solo todos los átomos que son demostrables. Formalmente $M$ es un modelo de un programa básico $\Pi$ si y solo si $M$ es el subconjunto minimal tal que por cada regla $p \leftarrow q_1, q_2, \ldots, q_n \in \Pi$ tal que $\{q_1, \ldots, q_n\} \subseteq M$, entonces $p \in M$.

Un elemento sintáctico importante relevante a esta tesis es la *negación como falla*. Las reglas que contienen estos átomos negados se ven como se muestra a continuación:

$$p \leftarrow q_1, q_2, \ldots, q_n, not\ r_1, not\ r_2, \ldots, not\ r_k. \tag{2.2}$$

Intuitivamente, la regla (2.2) debe ser interpretada como '$p$ es demostrable si $q_1, \ldots q_n$ son demostrables y ningún elemento de $r_1, \ldots, r_k$ es demostrable'. La semántica de los programas que incluyen la negación como falla es simple pero requiere la introducción de los llamados *modelos estables*. Omitiremos la definición formal de estos por temas de relevancia (Ferraris y Lifschitz, 2005).

Otro tipo de regla relevante es:

$$|\{p_1, p_2, \ldots, p_n\}| = k \leftarrow q_1, q_2, \ldots, q_n.$$

Intuitivamente decimos que si todos los elementos en $q_1, q_2, \ldots, q_n$ son demostrables, entonces $k$ de los elementos en $\{p_1, p_2, \ldots, p_n\}$ deben aparecer en el modelo. Esta definición permite a los programas tener multiples modelos[2]. Por ejemplo, el programa $\{s, |p, q, r| = 1 \leftarrow s\}$ tiene tres modelos: $\{p, s\}$, $\{q, s\}$, y $\{r, s\}$.

Otro tipo de regla relevante es:

$$\leftarrow p_1, p_2, \ldots, p_n \tag{2.3}$$

La cual es una restricción que prohíbe la aparición de $\{p_1, p_2, \ldots, p_n\}$ en el modelo.

Finalmente, los programas en ASP pueden contener variables para representar reglas con esquemas. Así, una regla como:

$$p(X) \leftarrow q(X) \tag{2.4}$$

Donde los caracteres en mayúscula representan variables. Intuitivamente una variable que aparece en un programa $\Pi$ puede tomar cualquier valor que se encuentre en el conjunto de *términos* de $\Pi$. Así, la regla (2.4) representa que cuando $c$ es un término y $q(c)$ es demostrable, entonces $p(c)$ también lo es. Un término representa un objeto que puede ser nombrado/identificado en el programa. El conjunto de términos para un programa está

[2]Técnicamente, la negación como falla sola también permite al usuario crear programas con múltiples modelos, pero en nuestra codificación definimos múltiples modelos explotando este tipo de reglas.

sintácticamente determinado usando las constantes mencionadas en él. El conjunto de términos tiene una contra parte teórica, la llamada *base de Herbrand*, cuya definición omitiremos, dado que no es relevante para la comprensión de este trabajo.

En el proceso de búsqueda de un modelo para un programa, un paso inicial que es llevado a cabo es el proceso de *instanciación*. Este proceso instancia las reglas que contienen variables para que estás sean removidas del programa. Dado que existen variadas técnicas optimización que los solvers utilizan durante la instanciación, no es fácil describir el proceso con precisión aquí, y por lo tanto solo lo describiremos de forma intuitiva. Por ejemplo, el siguiente programa:

$$\{q(a), q(b), p(X) \leftarrow q(X)\} \tag{2.5}$$

Puede ser transformado a:

$$\{q(a), q(b), p(a) \leftarrow q(a), p(b) \leftarrow q(b)\} \text{ o a}$$
$$\{q(a), q(b), p(a), p(b)\},$$

Dependiendo de las optimizaciones aplicadas en la instanciación. Lo que sí es inevitable es que el proceso genere dos instancias para la regla $p(X) \leftarrow q(X)$ debido a que el número de objetos que satisfacen el predicado $q$ es dos. Por lo tanto, si hubiéramos declarado $n$ objetos satisfaciendo $q$, esperaríamos que el proceso genere $n$ instancias de $p(X) \leftarrow q(X)$. Como veremos en las próximas secciones, el tamaño del programa luego del proceso de instanciación es clave en el rendimiento.

Finalmente, una característica muy relevante de ASP es que los programas también pueden contener una declaración de minimización (o de maximización) que exprese que se busca un modelo que minimice un determinado criterio. Estas son de la forma $minA$ (o $maxA$) e indican al solver que minimice (o maximice) el número de átomos en $A$ que aparecen en el modelo.

## 2.5. Optimizadores

Durante nuestros experimentos trabajamos con Clingo 5.3 como nuestra implementación de ASP, este posee distintas opciones que permiten ajustar la resolución según el programa que se quiera optimizar. Una de estas opciones es la elección del tipo de optimizador con que se resolverá el programa. En esta sección explicaremos brevemente los dos que utilizamos y como se diferencian.

### 2.5.1. Branch-and-Bound

*Branch-and-Bound* (BB) (Gebser, Kaminski, Kaufmann, y Schaub, 2011) es un método bastante conocido de optimización. El algoritmo inicia buscando una solución que satisfaga las reglas y restricciones del programa, luego usando el costo de esta solución poda la búsqueda hasta encontrar y demostrar una solución de costo mínimo.

### 2.5.2. Unsatisfiable Core

*Unsatisfiable Core* (USC) (Andres, Kaufmann, Matheis, y Schaub, 2012) por otro lado inicia buscando una solución de costo mínima que no necesariamente satisfaga las restricciones del programa en ASP. Si es que el programa no es resolvible utilizando como límite los costos dados por esta primera solución, entonces, USC intentará buscar incrementalmente soluciones de costo mayor hasta encontrar una que satisfaga las restricciones.

# 3. CODIFICACIÓN

En esta sección explicaremos la parte más importante del trabajo de esta tesis. A continuación, describiremos nuestra compilación de MAPF en su versión de suma de costos a un programa en ASP. Como mencionamos anteriormente está es la primera compilación de ASP que maneja la optimización bajo suma de costos.

## 3.1. Compilación Básica

Primero mostraremos una compilación básica que es similar en varios aspectos a la compilación de Erdem y cols. (Erdem y cols., 2013) en ASP. Y además similar en algunos aspectos a la compilación en SAT de Surynek (Surynek, 2014):

Como la mayoría de las compilaciones de problemas de planning a SAT/ASP, el makespan de la compilación es entregado como parámetro, el cual de ahora en adelante llamaremos `T`.

### 3.1.1. Átomos

Para nuestra codificación definimos los siguientes átomos:

- $agent(a)$: indica que $a$ es un agente,
- $goal(a, x, y)$: indica que la celda objetivo para el agente $a$ es $(x, y)$,
- $obstacle(x, y)$: indica que la celda $(x, y)$ is un obstáculo,
- $at(a, x, y, t)$: indica que el agente $a$ está en $(x, y)$ en tiempo $t$,
- $exec(a, m, t)$: indica que el agente $a$ ejecuta $m$ en tiempo $t$,
- $at_goal(a, t)$: indica que el agente $a$ está en su objetivo en tiempo $t$,
- $time(t)$: $t$ es un instante de tiempo,
- $move(m)$: $m$ es un movimiento.

Finalmente, usamos los átomos $rangeX(x)$ y $rangeY(y)$ para especificar qué $(x, y)$ está dentro de los límites de la grilla.

### 3.1.2. Especificación de la Instancia

Para especificar una instancia particular de MAPF, definimos como hechos los átomos de la forma $agent(a)$, por cada agente $a \in \mathcal{A}$. $obstacle(x, y)$ por cada celda $(x, y)$ que está marcada como un obstáculo en la grilla. $rangeX(x)$ para cada $x \in \{1, \ldots, w\}$, donde $w$ es el ancho de la grilla, y de forma similar $rangeY(y)$ por cada $y \in \{1, \ldots, h\}$, donde $h$ is el alto de la grilla. Adicionalmente, definimos la celdas iniciales de cada agente, agregando como hechos los átomos $at(a, x_a, y_a, 0)$ por cada uno de los agentes $a \in \mathcal{A}$, donde $(x_a, y_a) = init(a)$. Además, agregamos un átomo de la forma $time(t)$ por cada $t \in \{1, \ldots, \mathtt{T}\}$. Notamos que el número de reglas necesarias para codificar una instancia de MAPF por lo tanto está en $\Theta(|\mathcal{A}| + \mathtt{T} + |V|)$.

### 3.1.3. Efectos

Para codificar los efectos de las 5 acciones, usamos una regla escrita como sigue:

$$\begin{aligned} at(A, X, Y, T) \leftarrow & exec(A, M, T-1), \\ & at(A, X', Y', T-1), \\ & delta(M, X', Y', X, Y). \end{aligned} \tag{3.1}$$

La cual especifica que si el agente $A$ está en la posición $(X', Y')$ en el instante $T - 1$, entonces estará en la posición $(X, Y)$ en el instante $T$, si y solo si, $(X, Y)$ y $(X', Y')$ satisface el predicado $delta$. El predicado auxiliar $delta$ es usado para establecer la relación entre $(X, Y)$ and $(X', Y')$ dado un movimiento $M$ de la siguiente manera:

$$
\begin{aligned}
&delta(right, X, Y, X+1, Y) \leftarrow rangeX(X), rangeY(Y),\\
&delta(left, X, Y, X-1, Y) \leftarrow rangeX(X), rangeY(Y),\\
&delta(up, X, Y, X, Y+1) \leftarrow rangeX(X), rangeY(Y),\\
&delta(down, X, Y, X, Y-1) \leftarrow rangeX(X), rangeY(Y),\\
&delta(wait, X, Y, X, Y) \leftarrow rangeX(X), rangeY(Y).
\end{aligned} \tag{3.2}
$$

El predicado $delta$ generado en el proceso de instanciación resulta en 5 reglas por cada posición de la grilla. Esto define que el número total de instancias generadas para la regla (3.1) es proporcional al tamaño de la grilla, el número de agentes y el número de instantes de tiempo. El número total de instancias para las reglas de la forma (3.1) y (3.2) se encuentra en $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$.

### 3.1.4. Ejecución de movimientos en paralelo

Necesitamos codificar que cada agente realiza exactamente un movimiento por cada instante de tiempo. Para esto definimos la siguiente regla:

$$
|\{exec(A, M, T-1) : move(M)\}| = 1 \leftarrow time(T), agent(A). \tag{3.3}
$$

Luego del proceso de instanciación el número de instancias de esta regla está en $\Theta(|\mathcal{A}| \cdot \mathtt{T})$.

### 3.1.5. Posiciones legales

Necesitamos expresar que los agentes se mueven a través de los vértices en el grafo; es decir, no pueden salir de la grilla ni ingresar a una celda obstáculo. Hacemos esto usando las siguientes tres reglas:

$$
\begin{aligned}
&\leftarrow at(A, X, Y, T), not\ rangeX(X), \\
&\leftarrow at(A, X, Y, T), not\ rangeY(Y), \\
&\leftarrow at(A, X, Y, T), obstacle(X, Y).
\end{aligned}
\tag{3.4}
$$

El número todal de reglas generadas por las reglas en (3.4) está en $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$, dado que depende del número de átomos de la forma $at$, $obstacle$, $rangeX$, and $rangeY$.

### 3.1.6. Conflictos de Vértices

Para expresar que ningún par de agentes estén en la misma celda en una instancia de tiempo, usamos la siguiente restricción, la cual es similar a las codificación a ASP de Erdem y cols.; Gebser y cols. (Erdem y cols., 2013; Gebser y cols., 2018) y Surynek y cols. (Surynek y cols., 2016).

$$
\leftarrow at(A, X, Y, T), at(A', X, Y, T) \tag{3.5}
$$

El número de instancias para esta regla (3.5) luego de realizar el proceso de instanciación está en $\Theta(|\mathcal{A}|^2 \cdot |V| \cdot \mathtt{T})$. Notar que, hasta aquí, está es la primera regla en que su instanciación es cuadrática en el número de agentes. Esto motiva las mejorar que mostraremos en las siguientes secciones

### 3.1.7. Conflictos de intercambio de agentes

Ningún par de agentes pueden intercambiar celdas. Esto lo expresamos con dos restricciones que evitan intercambios horizontales y verticales

$$\begin{aligned}
&\leftarrow at(A, X+1, Y, T-1), at(A', X, Y, T-1),\\
&\quad at(A, X, Y, T), at(A', X+1, Y, T). \text{ \% Intercambio horizontal}\\
&\leftarrow at(A, X, Y+1, T-1), at(A', X, Y, T-1),\\
&\quad at(A, X, Y, T), at(A', X, Y+1, T). \text{ \% Intercambio vertical}
\end{aligned} \tag{3.6}$$

El número de reglas instanciadas para (3.6) está en $\Theta(|\mathcal{A}|^2 \cdot |V| \cdot \mathtt{T})$.

### 3.1.8. Logro del Objetivo

Especificamos con una restricción que ningún agente está fuera de su objetivo en el instante de tiempo T:

$$\begin{aligned}
&at_goal(A, T) \leftarrow at(A, X, Y, T), goal(A, X, Y),\\
&\leftarrow agent(A), not\ at_goal(A, \mathtt{T}).
\end{aligned} \tag{3.7}$$

El número de reglas instanciadas para 3.7 está $\Theta(\mathcal{A} \cdot |V| \cdot \mathtt{T})$.

### 3.1.9. Tamaño de la codificación Básica

Luego del proceso de instanciación, podemos ver que el tamaño de la codificación está en $\Theta(|\mathcal{A}|^2 \cdot |V| \cdot \mathtt{T})$. Es decir, es cuadrático en el número de agentes, lineal en el tamaño de la grilla, y lineal en el parámetro makespan T.

## 3.2. Suma de costos en ASP

La codificación presentada en la sección anterior aún no produce soluciones de costo óptimo. Si es que entregamos esta a un programa, ASP retornará un modelo si una solución

con makespan T existe. En esta sección explicaremos como podemos obtener soluciones que minimicen la función de suma de costos en MAPF.

Hay una forma natural e intuitiva de codificar la minimización, la cual es minimizar el número de acciones realizadas por cada agente antes de detenerse en su objetivo. Sin embargo, notamos que esto produce una instanciación que crece linealmente en el tamaño de la grilla, $|V|$. Esto nos motivó a buscar una codificación más compacta que no dependiera de $|V|$. Si bien, como mostraremos en las secciones posteriores, la codificación independiente de la grilla se desempeña mejor en la práctica, describiremos ambos enfoques, dado que la codificación dependiente de la grilla es más natural y es una contribución por si misma dado que la suma de costos no ha sido anteriormente definida en ASP.

### 3.2.1. Codificación dependiente de la grilla

Esta codificación es similar a la usada en MDD-SAT (Surynek y cols., 2016): La idea es minimizar las acciones realizadas por cada agente. A primera vista, uno tendería a pensar que solo necesitamos contar el número de acciones realizadas fuera de una celda objetivo y minimizar este número. Sin embargo, este enfoque no funciona debido a que realizar un $esperar$ en el objetivo en un instante $t$ debería ser contado si es que el agente se mueve fuera del objetivo en un instante $t'$ mayor a $t$.

Para identificar los instantes de tiempo en los que sabemos que el agente no se moverá fuera del objetivo, introducimos el predicado $at_goal_back(a, t)$, que indica que el agente $a$ a llegado a su objetivo en el instante $t$ y no se moverá en el futuro:

$$
\begin{aligned}
at_goal_back(A, \mathtt{T}) &\leftarrow agent(A), \\
at_goal_back(A, T-1) &\leftarrow at_goal_back(A, T), \\
& \quad\; exec(A, wait, T-1).
\end{aligned}
$$

Ahora definiremos el predicado $cost$, tal que hay un átomo de la forma $cost(a, t, 1)$ en el modelo cada vez que un agente $a$ realiza una acción en el instante $t$ antes de detenerse en su objetivo. Primero expresamos que mover un agente desde una celda que no es su objetivo es penalizado en una unidad:

$$cost(A, T, 1) \leftarrow at(A, X, Y, T), not\ goal(A, X, Y).$$

Segundo, mover un agente hacia afuera de su objetivo también es penalizado en una unidad:

$$\begin{aligned} cost(A, T, 1) \leftarrow & at(A, X, Y, t), goal(A, X, Y), \\ & exec(A, M, t), M \neq esperar. \end{aligned}$$

Tercero, si un agente ejecuta un $esperar$ en el objetivo, pero luego se mueva en un instante posterior, entonces también es penalizado:

$$\begin{aligned} cost(A, T, 1) \leftarrow & at(A, X, Y, t), goal(A, X, Y), \\ & exec(A, wait, T), not\ at_goal_back(A, T). \end{aligned}$$

Finalmente, bajo una declaración de optimización, minimizamos el número de átomos de la forma $cost(A, T, 1)$ en el modelo:

$$\#minimize\{C, T, A : cost(A, T, C)\}.$$

Luego de realizar el proceso de instanciación el número de reglas está en $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$.

### 3.2.2. Codificación independiente de la grilla

Para esta codificación, definimos el átomo $optimal(a, c_a)$, por cada agente $a \in \mathcal{A}$, donde $c_a$ corresponde al costo del camino óptimo desde $init(a)$ hasta $goal(a)$ ignorando todos los conflictos de vértices e intercambio entre agentes. En otras palabras, $c_a$ es el

resultado de resolver la relajación del problema que ignora los otros agentes. Computamos estos valores usando el algoritmo de BFS, previo a generar la codificación.

A diferencia de la codificación anterior, aquí maximizamos el rango entre el makespan $T$ y el instante de tiempo en que el agente se detiene en su objetivo, simplemente agregando:

$$\begin{aligned} penalty(A, T, 1) \leftarrow & optimal(A, C), T > C, \\ & at_goal_back(A, T - 1). \end{aligned}$$

Finalmente, definimos la siguiente declaración de optimización:

$$\#maximize\{P, T, A : penalty(A, T, P)\}.$$

Notar que debido a que no se hacen referencias a las celdas de la grilla, el proceso de instanciación genera un número de reglas en $\Theta(|\mathcal{A}| \cdot \mathtt{T})$.

### 3.3. Una Codificación Lineal

La codificación que hemos propuesto es cuadrática en el número de agentes. En esa sección mostraremos como linealizarla introduciendo nuevos átomos a la codificación. Específicamente introducimos los siguientes átomos:

- $rt(x, y, t)$ (resp. $lt(x, y, t)$) que especifica que la arista entre $(x, y)$ y $(x + 1, y)$ fue cruzada por algún agente en el instante $t$ de izquierda a derecha (resp. de derecha a izquierda).
- $ut(x, y, t)$ (resp. $dt(x, y, t)$) que especifica que la arista entre $(x, y)$ y $(x, y + 1)$ fue cruzada por algún agente en el instante $t$ hacia arriba (resp. hacia abajo).

- $st(x, y, t)$, indica que algún agente se quede $(x, y)$, esto significa que realizó una acción $esperar$ en el instante $t$.

Las dinámicas de estos átomos están definidas usando una regla con variables. Esta regla para $rt$ es:

$$rt(X, Y, T) \leftarrow exec(A, derecha, T), at(A, X, Y, T), \tag{3.8}$$

Mientras que la regla para $st$ es:

$$st(X, Y, T) \leftarrow at(A, X, Y, T), exec(A, esperar, T).$$

Omitimos las reglas $dt$, $lt$, and $ut$ dado a que son análogas a (3.8).

Usando estos predicados ahora podemos expresar el hecho que una celda no puede ser ingresada en el mismo instante de tiempo por dos agentes distintos. Esto requiere seis reglas que corresponden a todas las combinaciones de pares de acciones en $\{derecha, izquierda, arriba, abajo\}$. Por ejemplo, la siguiente regla expresa que la celda $(X, Y)$ no puede ser ingresada por un agente que realiza una acción $abajo$ al mismo tiempo que otro agente está ingresando a ella realizando una acción $arriba$:

$$\leftarrow lt(X, Y, T), dt(X, Y, T)$$

Es fácil verificar que estas reglas no mencionan pares de distintos agentes, en contraste a (3.5) y (3.6), por lo que luego de hacer la instanciación terminamos con número en $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$ de reglas y por lo tanto nuestra codificación es lineal en $|\mathcal{A}|$.

Cabe destacar que este cambio significa agregar un número constante de restricciones por cada instante de tiempo y agente. Ya que se debe verificar si es que su último movimiento hace una colisión con otro agente. Sin embargo, en nuestras pruebas nunca observamos un mejor rendimiento por parte de la codificación cuadrática. Potencialmente puede que existan instancias en las cuales la codificación lineal sea peor, pero estas son suficientemente simples y son opacadas por otros factores en la resolución.

### 3.4. Utilizando búsqueda para reducir el número de átomos

Podemos aprovechar nuestra ejecución de BFS durante el preproceso para generar una codificación aún más pequeña remplazando la regla (3.1) por la siguiente regla.

$$\begin{aligned} at(A, X, Y, T) \leftarrow & at(A, X', Y', T-1), exec(A, M, T), \\ & delta(M, X', Y', X, Y), \\ & cost_to_go(A, X, Y, C), T + C <= \texttt{T}, \end{aligned} \tag{3.9}$$

Donde $cost_to_go(A, X, Y, C)$ indica que $C$ es el mínimo número de acciones necesarias para ir desde $(X, Y)$ hasta el objetivo del agente $A$. De esta forma podemos ignorar la generación de reglas para posiciones que no alcanzaran el objetivo dada la limitante del makespan, generando así una codificación mucho más compacta. Esta idea está bastantemente relacionada con el uso de grafos MDDs en MDD-SAT (Surynek y cols., 2016), pero no requiere la generación del MDD, por lo que es conceptualmente más simple.

### 3.5. Encontrando Soluciones Óptimas

La codificación propuesta hasta ahora solo encuentra la solución costo-óptima dado cierto makespan. Todavía es necesario definir como encontrar la verdadera solución costo-óptima.

Siguiendo el enfoque usado por la codificaciones SAT para planeamiento (Kautz y Selman, 1992), en nuestro enfoque intentamos resolver las instancias incrementando el makespan $\mathtt{T}$, hasta que una solución, que llamaremos $sol_{opt}$, es encontrada. En este proceso dos puntos son importantes de mencionar.

Primero, no es necesario incrementar $\mathtt{T}$ desde 1. Como fue mencionado antes, cuando se hace el preproceso, para cada uno de los agentes computamos el costo $c_a^*$ que ignora el resto de los agentes. El makespan de cualquier solución debe ser al menos $\max_{a \in \mathcal{A}} c_a^*$, por lo que este puede ser la cota inferior de $\mathtt{T}$.

Segundo, sea $sol_{min}$ la primera solución encontrada. Lamentablemente, $sol_{min}$ es una solución makespan-óptima pero no necesariamente una solución costo-óptima. Sin embargo, utilizando esta solución, es posible computar una cota superior para el makespan más grande $\mathtt{T}_{max}$ que garantiza encontrar una solución costo-óptima. Todo esto basado en los resultados teóricos propuestos por Surynek y cols. (Surynek y cols., 2016):

**Teorema 3.1** (Surynek y cols. (2016))**.** *Sea $sol_{min}$ la solución makespan-óptima del problema $P$ en MAPF, sea $sol^-$ una solución costo-óptima de $P$ que ignora todos los conflictos, y sea $\mathtt{T}^-$ su respectivo makespan. Entonces el makespan de la solución costo-óptima es a lo más $\mathtt{T}_{max} = \mathtt{T}^- + c(sol_{min}) - c(sol^-) - 1$.*

DEMOSTRACIÓN. Sea $sol_{min}$ una solución makespan-óptima para el problema $P$, tal que no exista otra solución con el mismo makespan y menor costo. Sea $sol^-$ una solución para $P$ que ignora todos los conflictos entre los agentes. Para esta solución, $c(sol^-) = \sum_{a \in A} c_a^*$. Además, sea $\mathtt{T}^- = \max_{a \in \mathcal{A}} c_a^*$ el makespan de esta solución, y sea $sol_{opt}$ una solución óptima para $P$ en la función de suma de costos, donde $\mathtt{T}_{opt}$ es su respectivo makespan.

Debido a que el makespan siempre es un número entero, probamos que $\mathtt{T}_{opt} < \mathtt{T}^- + c(sol_{min}) - c(sol^-)$, que es equivalente a la descrito en el teorema. Lo demostraremos por contradicción, asumimos que:

$$\mathtt{T}_{opt} \geq \mathtt{T}^{-} + c(sol_{min}) - c(sol^{-}) \tag{3.10}$$

Debido a que $\mathtt{T}_{opt}$ es el makespan de $sol_{opt}$, existe un agente, que llamaremos $a_i$, tal que $c(sol_{opt}(a_i)) = \mathtt{T}_{opt}$, por lo tanto:

$$c(sol_{opt}) = \mathtt{T}_{opt} + \sum_{a \in A \setminus a_i} c(sol_{opt}(a)) \tag{3.11}$$

Dado que $c_a^* \leq c(sol_{opt}(a))$, para todo $a \in A$:

$$c(sol_{opt}) \geq \mathtt{T}_{opt} + \sum_{a \in A \setminus a_i} c_a^* \tag{3.12}$$

Usamos (3.10) para obtener:

$$c(sol_{opt}) \geq \mathtt{T}^{-} + c(sol_{min}) - c(sol^{-}) + \sum_{a \in A \setminus a_i} c_a^* \tag{3.13}$$

Dado que $c(sol_{min}) > c(sol_{opt})$, substituimos utilizando (3.13) para escribir:

$$c(sol_{min}) \geq \mathtt{T}^{-} + c(sol_{min}) - c(sol^{-}) + \sum_{a \in A \setminus a_i} c_a^*. \tag{3.14}$$

Cancelando $c(sol_{min})$ en (3.14), obtenemos:

$$c(sol^{-}) > \mathtt{T}^{-} + \sum_{a \in A \setminus a_i} c_a^* = c(sol^{-}),$$

Lo cual es una contradicción. □

Finalmente, luego que encontramos la primera solución $sol_{min}$, ejecutamos nuevamente el solver con makespan $\mathtt{T}_{max}$ dado por el Teorema 3.1, como se especifica en el Algoritmo 4. Asumimos que la función SOLVE invoca un solver en ASP para el programa $P$ en ASP, y retorna un modelo, si es que este existe, y retorna `insatisfacible` en caso contrario. El enfoque implementado por el Algoritmo 4 fue recientemente evaluado por Barták y Svancara (Barták y Svancara, 2019) para su solver de MAPF basado en Picat.

**Algoritmo 4**: MAPF en ASP bajo suma de costos

**Input**: Una instancia de MAPF $I = (G, \mathcal{A}, init, goal)$

```
T⁻ ← máx_{a∈A} c*(a)
T ← T⁻
C⁻ ← Σ_{a∈A} c*(a)
while true do
    P ← Codificar I con makespan T
    M ← SOLVE P
    if M ≠ insatisfacible then
        C ← costo de la solución dado por M
        Δ ← C − C⁻ − 1 // límite dado por el Teorema. 3.1
        if T⁻ + Δ ≤ T then
            return solución dada por M
        else
            P ← Codificar I con makespan T⁻ + Δ
            M* ← SOLVE P
            return solución dada por M*
        end
    end
    T ← T + 1
end
```

# 4. EXPERIMENTOS

El objetivo de nuestra evaluación empírica fue comparar el rendimiento de distintas variantes de nuestra codificación contra representantes de solvers basados en búsqueda y en transformaciones. Comparamos con el solver basado en SAT: MDD-SAT (Surynek, 2014) (enc=mdd), y también SMT-CBS (Surynek, 2019) [1], y los solvers basados en búsqueda: EPEA* (Goldenberg y cols., 2014), y ICBS-h (Felner y cols., 2018). Nuestra evaluación está enfocada en grillas de tamaños relativamente pequeños dado que el tiempo utilizado en instanciación crece demasiado para grillas más grandes (por ejemplo, de $512 \times 512$), haciendo que nuestro enfoque sea poco práctico.

El análisis de la sección experimental puede separarse en tres partes:

1. Primero, analizamos el rendimiento de múltiples configuraciones de nuestro solver, comparando las distintas codificaciones mostradas en la sección anterior, alternativas de paralelización y dos optimizadores distintos para resolver MAPF.
2. Luego, analizamos el rendimiento de las soluciones makespan-óptimas, que entregan una alternativa que no garantiza optimalidad, pero posee un rendimiento substancialmente mejor.
3. Finalmente, experimentamos en grillas vacías de distinto tamaño, y analizamos el comportamiento de nuestro solver a medida que crece el tamaño de las grillas.

El código usado en nuestra implementación está escrito en Python 3.7, usamos Clingo 5.3 (Gebser, Kaminski, Kaufmann, y Schaub, 2014) como nuestro solver ASP. Clingo fue ejecutado con 4 threads en su modo paralelo (*parallel-mode*), y utilizando USC como estrategia de optimización (a menos que se indique lo contrario). Todos los algoritmos con los que nos comparamos fueron obtenidos de sus respectivos autores.

[1] Ambos disponibles públicamente

Todos nuestros experimentos fueron ejecutados en un computador con procesador Intel Core i5-3570K con 3.40GHz y con 8GB de memoria RAM. Para todas nuestras ejecuciones fijamos un tiempo límite de 5 minutos.

## 4.1. Experimentos sobre distintas configuraciones

En esta sección experimentamos sobre distintas configuraciones de ASP, con el objetivo de encontrar los parámetros adecuados para optimizar el rendimiento de nuestro solver. En particular, comparamos distintas codificaciones basadas en cada una de nuestras mejoras de la sección anterior:

- **ASP-basic:** es nuestra codificación más básica que usa resolución de conflictos cuadrática y el uso de penalizaciones dependientes de la grilla.
- **GI *(Grid Independent):*** se refiere al uso de penalizaciones independientes de la grilla. Esta variante está descrita en la sección 3.2.2.
- **LC *(Linear Conflicts):*** se refiere al uso de la codificación lineal. Está descrita en la sección 3.3.
- **CG *(Cost to Go):*** se refiere al uso del precomputo de los costos de los caminos. Esta descrita en la sección 3.4.

Luego, gracias a las opciones de configuración que ofrece Clingo, experimentamos sobre dos versiones de nuestro solver: Una que no utiliza paralelización y otra que permite trabajar sobre 4 threads. Finalmente probamos dos optimizadores distintos para la resolución de los programas: *Branch-and-Bound* (BB) y *Unsatisfiable Core Optimization* (USC).

### 4.1.1. Grillas de $N \times N$ con obstáculos aleatorios

Primero, experimentamos en grillas de $8 \times 8$ y $20 \times 20$ con $10\,\%$ de obstáculos posicionados aleatoriamente. Para las grillas de $8 \times 8$ (resp. $20 \times 20$) generamos 150 (resp. 160)

problemas con un número de agentes entre $\{4, \ldots, 18\}$ (resp. $\{20, 22, \ldots, 50\}$). Los resultados para las grillas de $8 \times 8$ son omitidos debido a que son muy similares a los de $20 \times 20$. La tasa de éxito, y número de problemas resueltos versus tiempo son mostrados en la Figura 4.1.

En las grillas de $8 \times 8$, a medida que el número de agentes aumenta todas nuestras codificaciones superan el rendimiento de los otros algoritmos en términos de tasa de éxito. También vemos que nuestras modificaciones a la codificación básica generan excelentes resultados. Observamos una diferencia importante entre las codificaciones dependientes e independientes de la grilla.

Para las grillas de $20 \times 20$, observamos que nuestra codificación lineal resuelve la mayoría de los problemas y superan considerablemente nuestras codificaciones cuadráticas. Sin embargo, en este caso no observamos un impacto importante en la codificación independiente de la grilla sobre la dependiente. En contraste, es interesante observar los beneficios del uso de la regla CG como una manera de generar una codificación más compacta, como se puede apreciar en la Figura 4.1 CG mejora sustancialmente el tiempo de resolución. De hecho, el tiempo promedio de resolución es $1{,}74$ veces más pequeño utilizando ASP-GI-LC-CG en comparación a ASP-GI-LC. Además, reportamos una disminución con factor $1{,}98$ en el tiempo promedio de ejecución del proceso de instanciación utilizando CG.

Para analizar la influencia del número de obstáculos en las grillas, experimentamos también en grillas de $20 \times 20$ con obstáculos generados aleatoriamente sobre 20 agentes. Evaluamos la mejor codificación según los experimentos mostrados anteriormente y los ejecutamos con 1 y 4 threads. Generamos 100 problemas variando el porcentaje de obstáculos entre $\{0, 5\,\%, 10\,\%, \ldots, 45\,\%\}$. La Figura 4.2 muestra estos resultados. Nuevamente, nuestra codificación en ASP supera el rendimiento de los otros algoritmos. Adicionalmente, no observamos diferencias significativas entre las variantes de 1 y 4 threads.

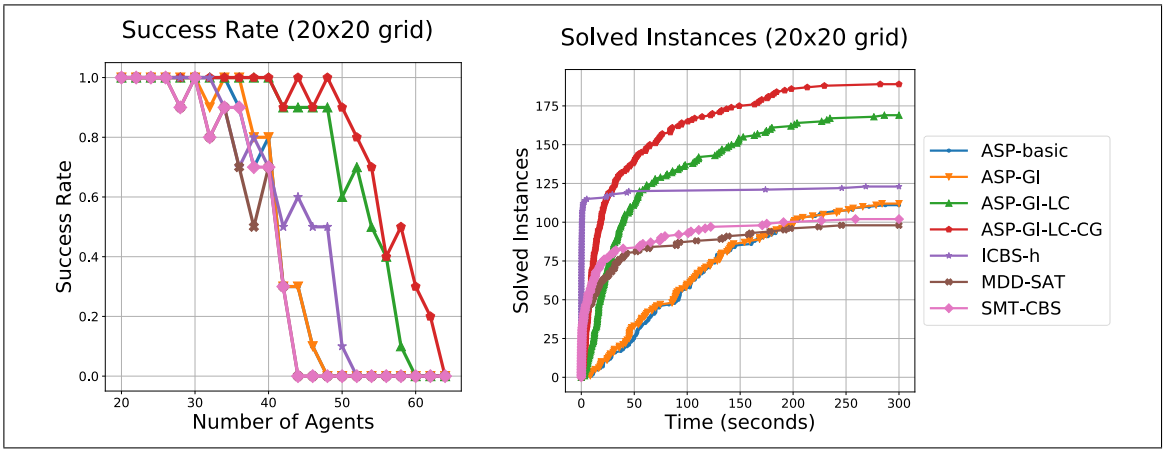



Figura 4.1. Tasa de éxito y número de instancias resueltas en grillas de $20 \times 20$ con distinto número de agentes. Notamos que ASP-GI-LC-CG es la mejor configuración, seguido de cerca por ASP-GI-LC con un rendimiento similar.

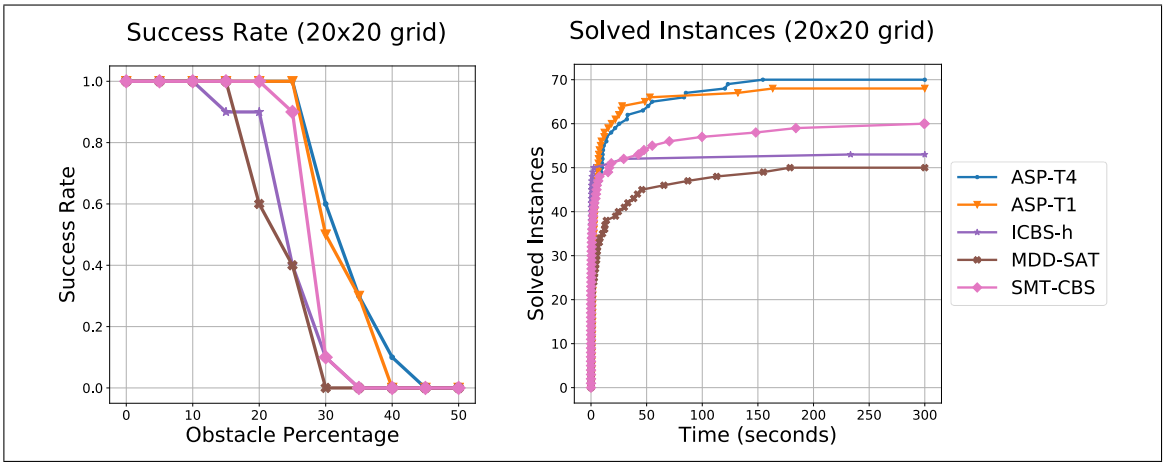



Figura 4.2. Tasa de éxito y número de instancias resueltas en grillas de $20 \times 20$ con distinto porcentaje de obstáculos. ASP-T1 es la mejor configuración (ASP-GI-LC-CG) ejecutada con 1 thread, y ASP-T4 es ASP-GI-LC-CG ejecutada con 4 threads. No vemos una diferencia importante entre las ejecuciones con distintos números de threads.

### 4.1.2. Experimentos en Bodegas

Experimentamos con una grilla de bodega normalmente utilizadas en la literatura de MAPF (por ejemplo, Felner y cols. (2018)), apreciada en la Figura 4.3. Seleccionamos de forma aleatoria posiciones iniciales y finales sobre los bordes izquierdos y derechos de las grillas. Generamos $10$ instancias por cada número de agentes en $\{4, \ldots, 16\}$.

En esta evaluación solo utilizamos la codificación con los mejores resultados de los experimentos sobre las grillas de $N \times N$ (ASP-GI-LC-CG). Además, probamos las dos estrategias de optimización ofrecidas por Clingo: Optimización basada en branch-and-bound (BB) (Gebser y cols., 2011) y optimización basada en unsatisfiable-core (USC) (Andres y cols., 2012). Las tasas de éxito y número de problemas resueltos versus tiempo son mostradas en la Figura 4.4.

Los resultados muestran los beneficios de nuestro enfoque sobre este tipo de grillas, donde nuestra codificación supera sustancialmente el rendimiento de los algoritmos con los que nos comparamos. Dado que el limitado número de celdas vacías en estas grillas, $|V|$ es más pequeño y por lo tanto nuestra codificación es bastante compacta. Estos resultados también ilustran que nuestro enfoque puede enfrentarse de forma más eficaz que otros enfoques cuando el número de potenciales conflictos aumenta.

En cuanto a USC versus BB. Observamos que USC indica un mejor rendimiento que BB para la resolución de MAPF costo-óptima. Recordamos que USC inicia con una solución de costo mínimo que no necesariamente cumple las condiciones de la codificación, y que, por otro lado, BB encuentra una solución inicial, para luego podar la búsqueda a partir de esta. Por lo que, considerando que en este tipo de problemas las soluciones normalmente son bastante cercanas a la solución de costo mínimo que ignora conflictos entre agentes es que se explica la diferencia de rendimiento.

También experimentamos en bodegas de tamaños $18 \times 21$, $27 \times 21$, $36 \times 21$, $9 \times 39$, $9 \times 57$, y $36 \times 57$. En la bodega más grande ($36 \times 57$) nuestro enfoque empieza a ser

ligeramente superado por ICBS-h, mientras que en el resto de las grillas, nuestro enfoque presenta una tendencia similar a la de la Figura 4.4.

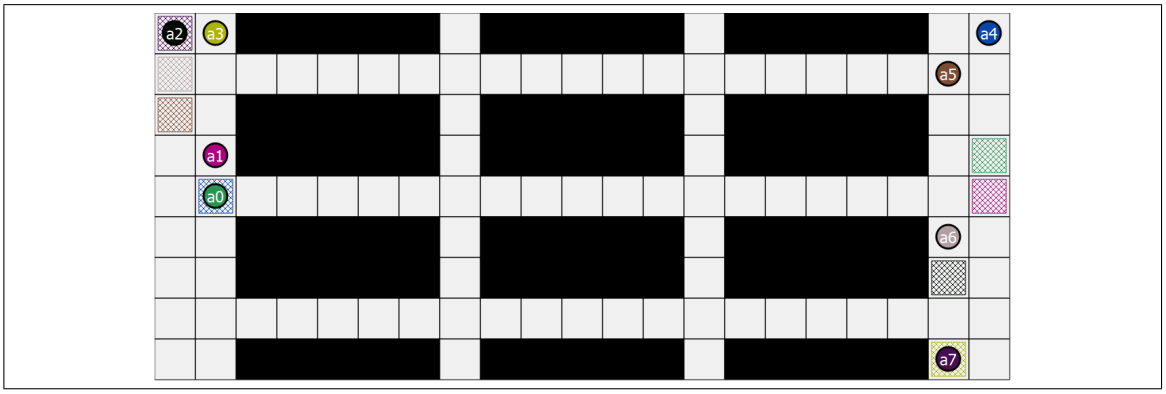



Figura 4.3. Ejemplo de problema de bodega.

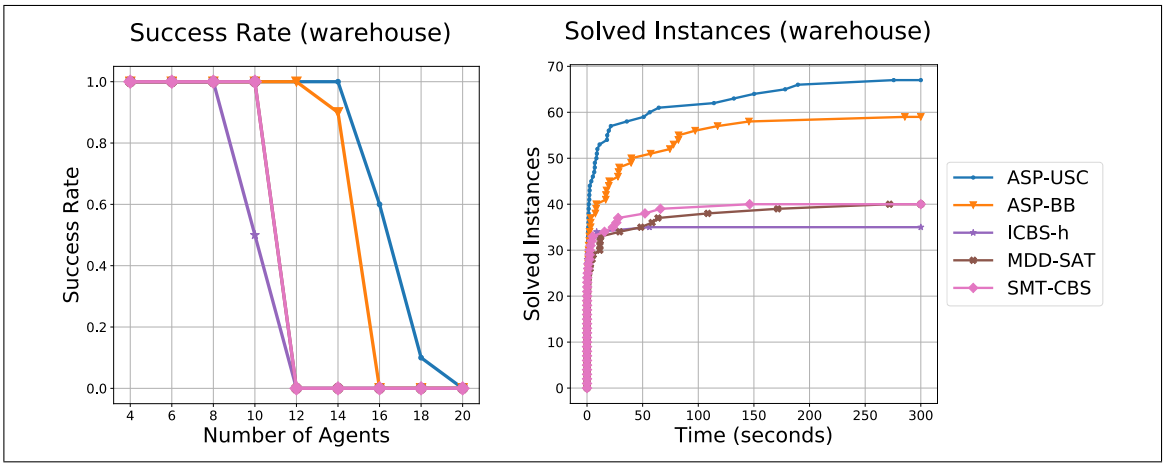



Figura 4.4. Tasa de éxito y número de instancias resueltas en mapa de bodega con distinto número de agentes. Apreciamos un mejor rendimiento por parte de USC.

## 4.2. Soluciones makespan-óptimas

Como vimos anteriormente, las soluciones makespan-óptimas pueden no ser costo-óptimas. Pero encontrar soluciones makespan-óptimas requiere menos tiempo. De hecho, el Algoritmo 4 ejecuta incrementalmente el solver ASP aumentando los valores del makespan hasta encontrar una primera solución. Después de eso, utiliza el límite del Teorema 3.1 para encontrar otra solución, con un mayor makespan, la cual está garantizada como costo-óptima. Debido a que esta última ejecución requiere el uso de un makespan más grande, la codificación también es mayor, lo que tiene un impacto en el tiempo de ejecución. Esto plantea una serie de preguntas sobre la diferencia entre encontrar soluciones makespan-óptimas y costo-óptimas. ¿Cuánto tiempo de resolución nos ahorraríamos si simplemente calculáramos soluciones makespan-óptimas en lugar de soluciones costo-óptimas? ¿Cuál es la suboptimidad de las soluciones makespan-optimas en comparación con las soluciones costo-óptimas? Intentamos responder a estas preguntas aquí.

Para el mismo conjunto de grillas de $20 \times 20$ de la sección anterior, ahora registramos el tiempo que necesario para encontrar la solución makespan-óptima, y su makespan. Primero nos centramos en la suboptimidad de las soluciones óptimas. Definimos el *porcentaje de suboptimalidad* como el porcentaje de la diferencia entre el costo de ambas soluciones con respecto al costo de la solución costo-óptima. En las Tablas 4.1 y 4.2 se muestra el porcentaje promedio de suboptimalidad, respectivamente, cuando se aumenta el número de agentes y cuando se aumenta el porcentaje de obstáculos. Por último, la Tabla 4.3 muestra el porcentaje promedio de suboptimalidad del mapa de bodega de la Figura 4.3 cuando se incrementa el número de agentes.

Observamos que el porcentaje promedio de suboptimalidad en general es muy bajo: para la mayoría de las configuraciones es inferior al 0.5 %, y su valor máximo es sólo 1.202 % en grillas $20 \times 20$ con un número relativamente alto de obstáculos. Esto significa que las soluciones makespan-óptimas y costo-óptimas son muy similares en cuanto a costos en general, y casi siempre equivalentes.

Tabla 4.1. Porcentaje promedio de suboptimalidad de las soluciones makespan-óptimas y número de instancias en las cuales la solución makespan-óptima y costo-óptima tienen el mismo costo, para grillas de $20 \times 20$ con $10\,\%$ de obstáculos variando el número de agentes. Notamos que la suboptimalidad es bastante baja en general y no se ve afectada por el número de agentes.

| **# agentes** | **Suboptimalidad promedio** | **Mismo costo %** |
|---:|---:|---:|
| 20 | 0,000 % | 100,0 % |
| 22 | 0,496 % | 80,0 % |
| 24 | 0,062 % | 90,0 % |
| 26 | 0,000 % | 100,0 % |
| 28 | 0,000 % | 100,0 % |
| 30 | 0,000 % | 100,0 % |
| 32 | 0,485 % | 90,0 % |
| 34 | 0,000 % | 100,0 % |
| 36 | 0,000 % | 100,0 % |
| 38 | 0,000 % | 100,0 % |
| 40 | 0,323 % | 80,0 % |
| 42 | 0,000 % | 100,0 % |
| 44 | 0,000 % | 100,0 % |
| 46 | 0,000 % | 100,0 % |
| 48 | 0,340 % | 80,0 % |
| 50 | 0,000 % | 100,0 % |
| 52 | 0,000 % | 100,0 % |
| 54 | 0,019 % | 85,7 % |
| 56 | 0,000 % | 100,0 % |
| 58 | 0,000 % | 100,0 % |
| 60 | 0,000 % | 100,0 % |
| 62 | 0,000 % | 100,0 % |
| 64 | – | – |
| 66 | 0,000 % | 100,0 % |
| 68 | 0,000 % | 100,0 % |
| 70 | – | – |

Tabla 4.2. Porcentaje promedio de suboptimalidad de las soluciones makespan-óptimas y número de instancias en las cuales la solución makespan-óptima y costo-óptima tienen el mismo costo, para grillas $20 \times 20$ con 20 agentes, en donde variamos el porcentaje de obstáculos. Notamos que la suboptimalidad es baja en general, pero viéndose afectada por el número de obstáculos.

| **% obstáculos** | **Suboptimalidad promedio** | **Mismo costo %** |
|---:|---:|---:|
| 0 | 0,000 % | 100,0 % |
| 5 | 0,112 % | 90,0 % |
| 10 | 0,000 % | 100,0 % |
| 15 | 0,359 % | 80,0 % |
| 20 | 0,575 % | 80,0 % |
| 25 | 0,238 % | 80,0 % |
| 30 | 0,801 % | 50,0 % |
| 35 | 1,668 % | 33,3 % |
| 40 | 0,909 % | 0,0 % |

Tabla 4.3. Porcentaje promedio de suboptimalidad de las soluciones makespan-óptimas y número de instancias en las cuales la solución makespan-óptima y costo-óptima tienen el mismo costo, en el mapa de bodega de la Figura 4.3. Notamos que la suboptimalidad es bastante baja en general y no se ve afectada por el número de agentes.

| **# agentes** | **Suboptimalidad promedio** | **Mismo costo %** |
|---:|---:|---:|
| 4 | 0,000 % | 100,0 % |
| 6 | 0,073 % | 90,0 % |
| 8 | 0,054 % | 90,0 % |
| 10 | 0,258 % | 80,0 % |
| 12 | 0,036 % | 90,0 % |
| 14 | 0,120 % | 80,0 % |
| 16 | 0,570 % | 66,7 % |
| 18 | 0,000 % | 100,0 % |

La siguiente pregunta que abordamos es cómo se ve afectado el rendimiento cuando configuramos el solver para encontrar soluciones makespan-óptimas en lugar de soluciones costo-óptimas. Las Figuras 4.5, 4.6, y 4.7 muestran el rendimiento de ASP-makespan, nuestra codificación en ASP configurada para devolver soluciones makespan-óptimas, frente a ASP-cost, nuestro solver costo-óptimo. Comparamos el rendimiento de ambos solvers en los mismos escenarios en los que hemos comparado anteriormente (bodegas con un número variable de agentes y grillas de $20 \times 20$ con un número variable de agentes y tasas de obstáculos).

Observamos que ASP-makespan escala sustancialmente mejor que ASP-cost cuando se aumenta el número de agentes o la tasa de obstáculos. En efecto, en el experimento sobre grillas de $20 \times 20$, cuando variamos el número de agentes, ASP-makespan resuelve 42 casos más que ASP-cost (un aumento del 21 %), en el experimento sobre grillas de $20 \times 20$, cuando variamos la tasa de obstáculos, ASP-makespan resuelve 31 casos más que ASP-cost (un aumento del 43 %) y, finalmente, en el experimento sobre bodegas, cuando variamos el número de agentes, ASP-makespan resuelve 21 casos más que ASP-cost (un aumento del 30 %).

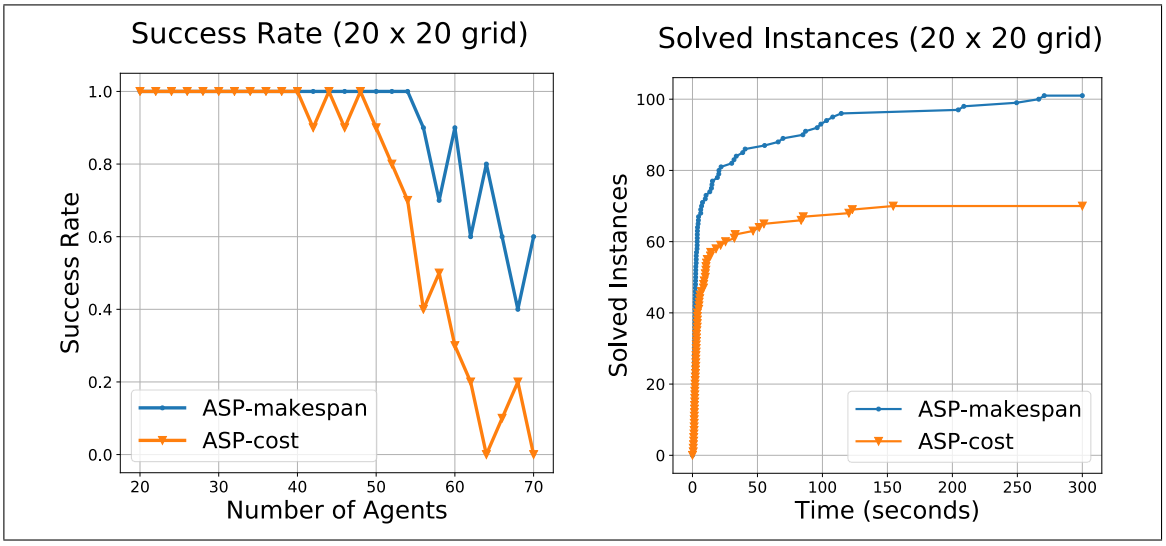



Figura 4.5. Tasa de éxito y número de instancias resueltas a medida que el número de agentes varía en una grilla de $20 \times 20$ con $10\,\%$ de obstáculos. ASP-makespan encuentra una solución makespan-óptima, mientras que ASP-cost encuentra una solución costo-óptima. Apreciamos un rendimiento substancialmente mejor por parte de ASP-makespan.

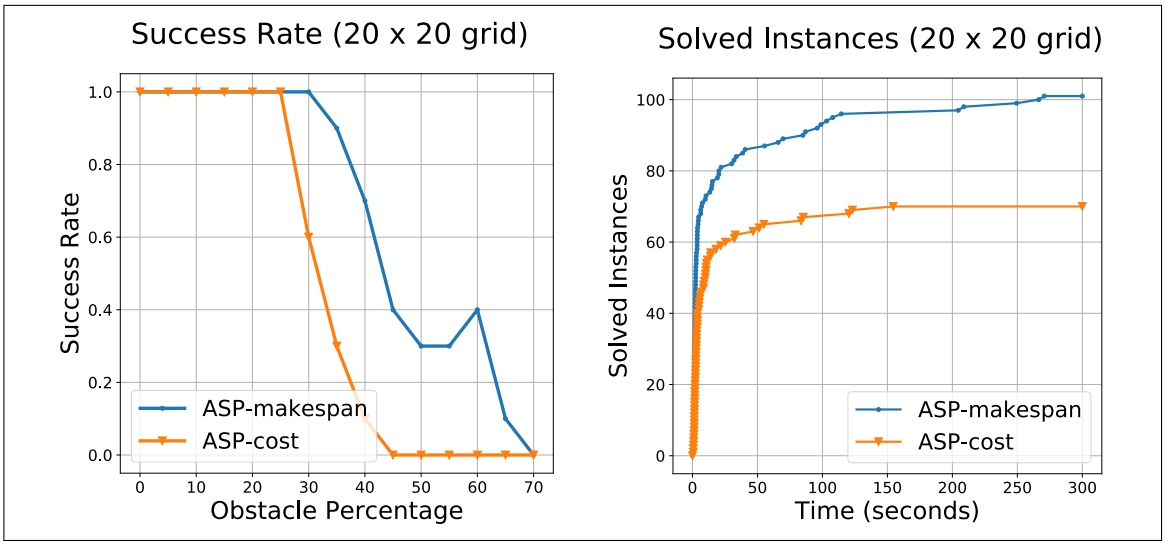



Figura 4.6. Tasa de éxito y número de instancias resueltas a medida que el porcentaje de obstáculos varía en una grilla de $20 \times 20$ con 20 agentes. ASP-makespan encuentra una solución makespan-óptima, mientras que ASP-cost encuentra una solución costo-óptima. Apreciamos un rendimiento substancialmente mejor por parte de ASP-makespan.

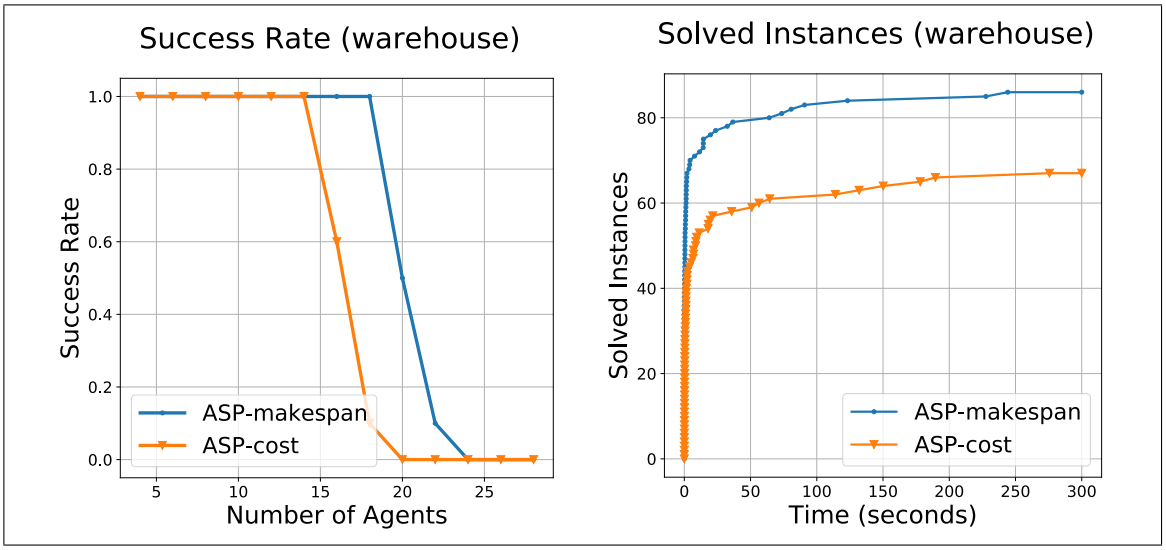



Figura 4.7. Tasa de éxito y número de instancias resueltas a medida que el número de agentes varía en la bodega de la Figura 4.3. ASP-makespan encuentra una solución makespan-óptima, mientras que ASP-cost encuentra una solución costo-óptima. Apreciamos un rendimiento substancialmente mejor por parte de ASP-makespan.

Tanto ASP-cost como ASP-makespan invocan al solver ASP incrementando iterativamente el makespan hasta que se encuentra una solución. Mientras que ASP-makespan retorna la primera solución encontrada, ASP-cost necesita ejecutar el solver una vez más sobre una codificación con makespan $\mathtt{T}^{-} + \Delta$ (Linea 4; Algoritmo 4). El valor $\Delta$ está dado por el límite teórico del Teorema 3.1.

Dado que ASP-cost es superado de manera significativa por ASP-makespan, una pregunta natural que surge es *cuán estrecho es ese límite*. Para responder a esta pregunta, definamos $\delta$ como $\mathtt{T}^{-} + \Delta - \mathtt{T}$, es decir, la diferencia entre el makespan de la última llamada y el utilizado en la penúltima llamada al solver ASP. Por lo tanto, intuitivamente, $\delta$ es un proxy del "esfuerzo extra" que el solver ASP necesita hacer para encontrar la última solución (recordar que la codificación es lineal en el tamaño del makespan). Además, definamos $\delta_{\text{oracle}}$ como la diferencia entre el makespan de la solución de costo óptimo y el makespan usado en la penúltima ejecución de ASP-cost. Es decir, intuitivamente, $\delta_{\text{oracle}}$ es el mínimo esfuerzo extra posible que ASP-cost tendría que hacer para encontrar la solución óptima si tuviéramos un oráculo para el makespan de la solución costo-óptima.

Las Tablas 4.4, 4.5, y 4.6 muestran el promedio de $\delta$ y $\delta_{\text{oracle}}$ para los experimentos sobre grillas y bodegas. Observamos que $\delta$ tiende a ser más alto que $\delta_{\text{oracle}}$. Además, la tasa de crecimiento de $\delta$ es más alta que la de $\delta_{\text{oracle}}$. Esto sugiere que hay un potencial significativo para la mejora de ASP-cost. Un factor clave de mejora vendría de encontrar un límite mejor que el dado por el Teorema 3.1.

Tabla 4.4. $\delta$ promedio y $\delta_{\text{oracle}}$ promedio, que son, respectivamente, proxies del esfuerzo actual e ideal necesario para encontrar la solución costo-óptima. El número de instancias resueltas también es mostrado como referencia. Experimentamos sobre grillas de $20 \times 20$ con $10\,\%$ de obstáculos y número creciente de agentes. Vemos un aumento significativo de $\delta$ en comparación a $\delta_{\text{oracle}}$.

| **# agentes** | **resueltos** | $\delta_{\text{oracle}}$ **promedio** | $\delta$ **promedio** |
|---:|---:|---:|---:|
| 20 | 10 | 0,0 | 0,0 |
| 22 | 10 | 0,3 | 2,0 |
| 24 | 10 | 0,2 | 2,0 |
| 26 | 10 | 0,0 | 1,5 |
| 28 | 10 | 0,0 | 3,5 |
| 30 | 10 | 0,0 | 4,0 |
| 32 | 10 | 0,2 | 4,0 |
| 34 | 10 | 0,0 | 6,0 |
| 36 | 10 | 0,0 | 9,0 |
| 38 | 10 | 0,0 | 4,0 |
| 40 | 10 | 0,4 | 7,5 |
| 42 | 9 | 0,1 | 9,0 |
| 44 | 10 | 0,1 | 7,5 |
| 46 | 9 | 0,0 | 11,0 |
| 48 | 10 | 0,4 | 13,0 |
| 50 | 9 | 0,0 | 17,0 |
| 52 | 8 | 0,1 | 12,5 |
| 54 | 7 | 0,1 | 16,0 |
| 56 | 4 | 0,5 | 12,0 |
| 58 | 5 | 0,0 | 17,0 |
| 60 | 3 | 0,0 | 16,0 |
| 62 | 2 | 0,0 | 23,0 |
| 64 | 0 | – | – |
| 66 | 1 | 0,0 | 26,0 |
| 68 | 2 | 0,0 | 24,0 |
| 70 | 0 | – | – |

Tabla 4.5. En grillas de $20 \times 20$ con 20 agentes y un porcentaje creciente de obstáculos, son mostrados el $\delta$ promedio y $\delta_{\text{oracle}}$ promedio, que son, respectivamente, proxies del esfuerzo actual e ideal necesario para encontrar la solución costo-óptima. El número de instancias resueltas también es mostrado como referencia. Vemos un aumento significativo de $\delta$ en comparación a $\delta_{\text{oracle}}$.

| **% obstáculos** | **resueltos** | $\delta_{\text{oracle}}$ **promedio** | $\delta$ **promedio** |
|---:|---:|---:|---:|
| 0 | 10 | 0,0 | 0,0 |
| 5 | 10 | 0,2 | 1,0 |
| 10 | 10 | 0,0 | 0,5 |
| 15 | 10 | 0,4 | 2,5 |
| 20 | 10 | 0,4 | 8,5 |
| 25 | 10 | 0,6 | 12,5 |
| 30 | 6 | 2,0 | 26,0 |
| 35 | 3 | 2,3 | 52,0 |
| 40 | 1 | 5,0 | 37,0 |

Tabla 4.6. En un mapa de bodega, con un número creciente de agentes, son mostrados el $\delta$ promedio y $\delta_{\text{oracle}}$ promedio, que son, respectivamente, proxies del esfuerzo actual e ideal necesario para encontrar la solución costo-óptima. El número de instancias resueltas también es mostrado como referencia. Vemos un aumento significativo de $\delta$ en comparación a $\delta_{\text{oracle}}$.

| **# agentes** | **resueltos** | $\delta_{\text{oracle}}$ **promedio** | $\delta$ **promedio** |
|---:|---:|---:|---:|
| 4 | 10 | 0,1 | 0,0 |
| 6 | 10 | 0,2 | 1,0 |
| 8 | 10 | 0,3 | 3,0 |
| 10 | 10 | 0,7 | 8,0 |
| 12 | 10 | 0,8 | 13,0 |
| 14 | 10 | 0,4 | 17,5 |
| 16 | 6 | 1,8 | 21,0 |
| 18 | 1 | 1,0 | 23,0 |

## 4.3. Grillas $N \times N$ Libres de Obstáculos

En los experimentos presentados anteriormente en grillas de $20 \times 20$, observamos que la tasa de éxito disminuye a medida que aumenta el número de agentes. De hecho, en la Figura 4.5 se puede observar que cuando resolvemos instancias con 58 agentes obtenemos una tasa de éxito del 50 %, pero cuando intentamos resolver problemas más grandes, obtenemos tasas de éxito más bajas. Por lo tanto, se podría argumentar que para $20 \times 20$, 58 agentes es el punto en el que los problemas empiezan a ser difíciles. En otras palabras, dado que el área una grilla de $20 \times 20$ es 400, decimos que los problemas se vuelven más difíciles cuando la *tasa de ocupación* es de alrededor de $58/400$ o 14.5 %.

Queríamos entender para qué tasa de ocupación observamos que la tasa de éxito cae por debajo del 50 % en función del tamaño de la red. Para ello diseñamos un nuevo conjunto de experimentos en los que usamos grillas libres de obstáculos de tamaño $N \times N$ con $N \in \{8, 16, 32\}$. Para las grillas de tamaño $8 \times 8$, fijamos el número de agentes en $\{2, 4, \ldots, N^2 - 2\}$, y generamos 10 problemas aleatorios. Para las grillas de $16 \times 16$ y $32 \times 32$, fijamos el número de agentes en $\{5, 10, \ldots, 150\}$ y generamos 10 problemas aleatorios.

Figura 4.8. Tasa de éxito de nuestro solver versus ICBS-h en grillas libres de obstáculos de distinto tamaño. Apreciamos una aparente caída del rendimiento por parte de ASP-cost a medida que aumenta el número de agentes.

La Figura 4.8 nos permite hacer dos observaciones.

1. El punto en el que la tasa de éxito cruza el 50 % es aproximadamente 31 agentes (48 % de ocupación) para $8 \times 8$, 66 agentes (25 % de ocupación) para $16 \times 16$ y 70 agentes (1.7 % de ocupación) para $32 \times 32$. Así, la tasa de ocupación en la cual los problemas se hacen más difíciles cae rápidamente a medida que el tamaño de las grillas aumenta.
2. En las grillas de $8 \times 8$ y $16 \times 16$, nuestro solver supera a ICBS-h, pero no lo hace en las grillas de $32 \times 32$.

Juntas estas dos observaciones pueden parecer indicar que nuestro solver no escala bien en grillas más grandes. Sostenemos que este no es necesariamente el caso. De hecho, lo que sucede es que los problemas se hacen mucho más difíciles cuando se aumenta el tamaño de las grillas y por lo tanto requieren más tiempo. La aparente caída en el rendimiento se debe por lo tanto al hecho de que no aumentamos el límite de tiempo de ejecución (establecido en 300 segundos para cada ejecución).

Para probar esta hipótesis, utilizamos nuestros datos de $20 \times 20$ y mostramos cómo se vería la tasa de éxito si el límite de tiempo de ejecución se hubiera fijado en 15 segundos (Figura 4.9). Observamos que dados sólo 15 segundos por instancia, ICBS-h supera a nuestro solver.

Otra prueba que sugiere que las grillas de $32 \times 32$ requieren más tiempo de ejecución por instancia proviene de la forma de las curvas de tiempo de ejecución promedio. La Figura 4.10 muestra el tiempo total de ejecución y el tiempo instanciación (es decir, el tiempo dedicado al proceso de instanciación) en una escala logarítmica. Tanto en las grillas de $8 \times 8$ y $16 \times 16$, utilizando una escala logarítmica, observamos que la curva de tiempo de ejecución es esencialmente una línea recta, mostrando que el tiempo de ejecución promedio crece exponencialmente con el número de agentes. Para las grillas de $32 \times 32$, las curvas se asemejan a un logaritmo.

Curiosamente, cuando limitamos el tiempo de ejecución a 15 segundos en nuestra grilla de $20 \times 20$ también obtenemos una curva de tipo logarítmico (Figura 4.11). Nuestras

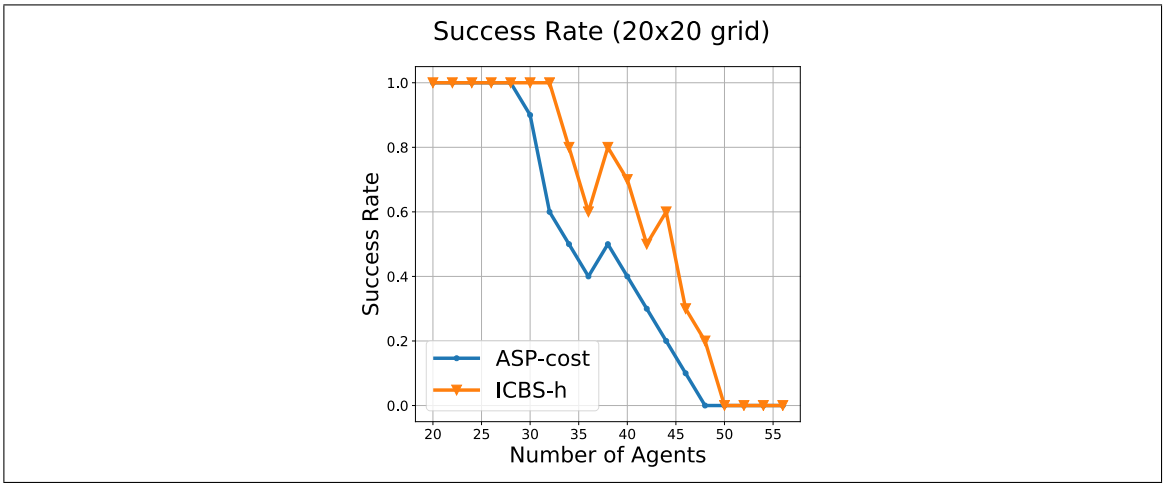



Figura 4.9. Tasa de éxito en una grilla de $20 \times 20$ con 10 % de obstáculos, donde el tiempo límite por instancia es fijado en 15 segundos, y el número de agentes varía. Vemos que cuando se disminuye el tiempo límite a 15 segundos, ICBS-h supera a ASP-cost.

codificaciones en ASP crecen linealmente con el número de agentes y el hecho de que el tiempo de ejecución crece exponencialmente es consistente con ese hecho. Pero como el tiempo de ejecución en nuestros experimentos de grillas $32 \times 32$ no muestra esta tendencia, concluimos que se necesita más tiempo de espera por cada instancia. Es necesario investigar más para establecer si en cualquier configuración se da el caso de que un aumento lineal en lugar de uno exponencial en el tiempo de ejecución se debe a un límite de tiempo establecido demasiado bajo.

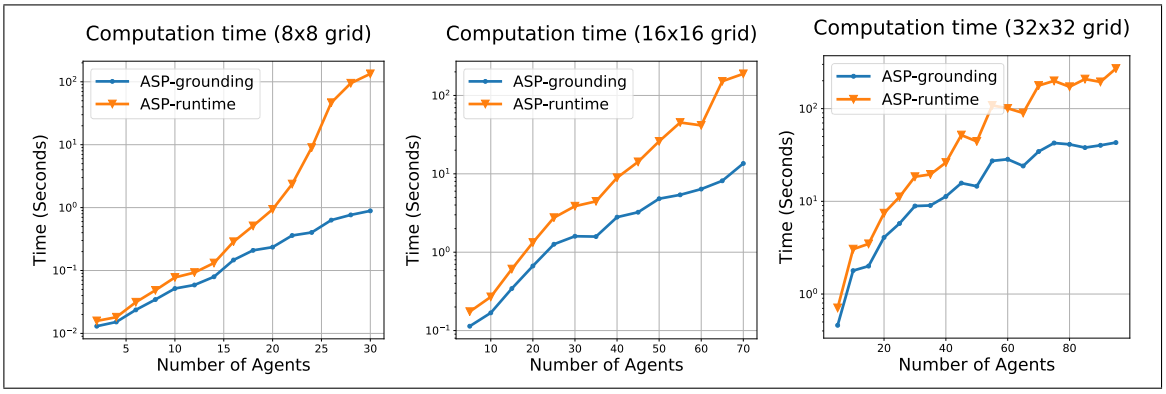



Figura 4.10. Tiempo total en escala logarítmica de ejecución e instanciación requerido para instancias en grillas libres de obstáculos de distinto tamaño. Observamos que las curvas de las grillas de $8 \times 8$ y $16 \times 16$ se asemejan a una recta, mientras que para $32 \times 32$ se asemejan a un logaritmo.

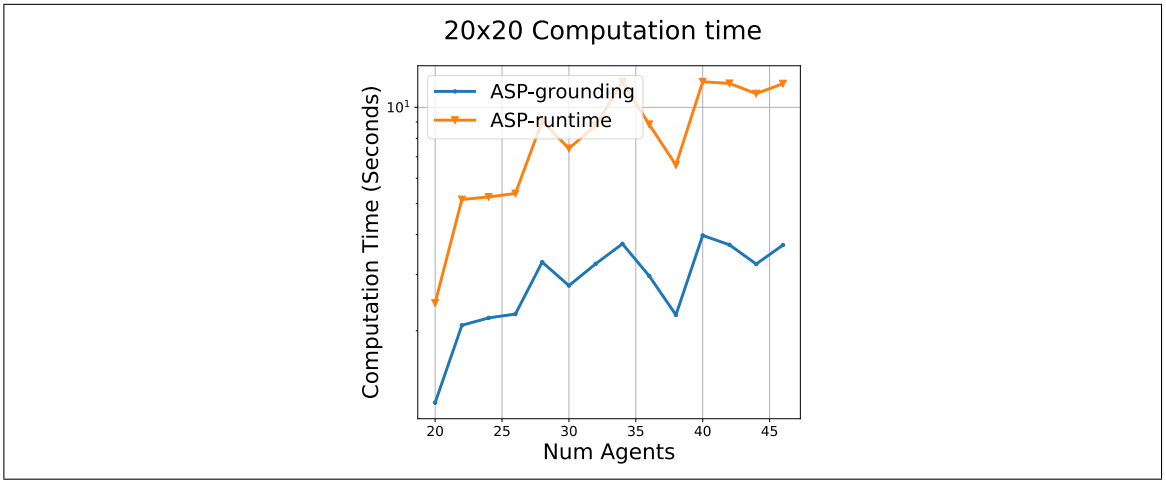



Figura 4.11. Tiempo total en escala logarítmica de ejecución e instanciación para grillas de $20 \times 20$ con $10\,\%$ de obstáculos, donde el tiempo límite por instancia es fijado en 15 segundos, y el número de agentes varía. Observamos que las curvas se asemejan a un logaritmo.

# 5. EXPANDIENDO ASP

Durante las secciones anteriores de esta tesis explicamos la construcción de nuestro solver basado en ASP que logra resolver MAPF óptimamente bajo la función de suma de costos. Donde, generamos una codificación de tamaño lineal en el número de agentes, que además aprovecha distintas optimizaciones, como el uso de búsqueda en preproceso para reducir el número de átomos, y una codificación inteligente del objetivo para agilizar el proceso de resolución.

Sin embargo, como se mostró en la última sección experimental, nuestra codificación tiene evidentes problemas de escalabilidad. Si bien, se logran resolver eficientemente instancias en mapas pequeños con alta congestión, incluso con un rendimiento por sobre el estado del arte. Nuestro solver no logra resolver mapas de mayor tamaño (por ejemplo $100 \times 100$), independiente de la dificultad de estos.

Por esta razón es interesante estudiar alternativas que permitan a nuestro solver solucionar estos problemas. En esta sección mostraremos una nueva versión de nuestro enfoque que permitirá manejar grillas de tamaños mayores sacrificando la optimalidad de las soluciones. La nueva versión hará uso de *ventanas* para simplificar el problema en secciones de tiempo, basándose en ideas propuestas por otro algoritmo en el estado del arte.

Como vimos anteriormente en la introducción de esta tesis, los solvers de MAPF pueden ser divididos en 2 clases distintas según como realizan su fase de planeamiento y ejecución:

## 5.1. Solvers Offline

Los solvers offline se caracterizan por primero realizar todo el planeamiento en un inicio, donde hacen todos los cómputos necesarios para resolver el problema y así luego, entregar un plan válido (en nuestro caso, la secuencia de caminos libres de conflictos de

los agentes). Luego de esto, los agentes se mueven según este plan sin realizar cómputos adicionales. Es decir, las fases de planeamiento y ejecución están totalmente separadas y solo se realizan una vez.

Todo los solvers mostrados hasta ahora en esta tesis han sido solvers offline, incluido nuestro solver propuesto anteriormente.

## 5.2. Solvers Online

Los solvers online, a diferencia de los offline, intercalan las fases de planeamiento y ejecución. Primero computan caminos no necesariamente completos para que luego en una fase de ejecución se muevan parcialmente según ese planeamiento. Desde las nuevas posiciones dada la ejecución anterior se realiza nuevamente una nueva fase de planeamiento y ejecución. Este proceso se va repitiendo sucesivamente hasta llegar a una solución.

Estos solvers sacrifican optimalidad y muchas veces no son completos, sin embargo, normalmente son más rápidos que su contra parte offline y pueden trabajar problemas más complejos. Son especialmente usados en ambientes donde los agentes necesitan moverse constantemente en intervalos de tiempo limitados.

## 5.3. WHCA*

WHCA* (*Windowed hierarchical cooperative A**  por sus siglas en inglés) (Silver, 2005) es un algoritmo online en el estado del arte que resuelve MAPF en su versión de suma de costos, que utiliza distintas técnicas para agilizar la búsqueda. Similar a CBS, realiza búsquedas independientes por cada agente usando A*, compartiendo una tabla de reservación entre éstas, con el fin de evitar conflictos. Además, mediante el uso de ventanas de tamaño fijo limita las fases de planeamiento y ejecución. Por último, aprovecha los cálculos hechos en fases anteriores, conservando información de los cálculos heurísticos hechos, para utilizarlos posteriormente y ahorrar tiempo de cómputo.

El algoritmo está descrito en Algortimo 5, recibe un problema en MAPF y un número entero `W` que indica el tamaño de las ventanas.

Primero, se define una tabla de reservación $\mathcal{T}$ que comparten los agentes, donde una reservación es una tupla $(a, v, t)$ que indica que el agente $a$ está en la posición $v$ en el instante $t$. Al inicio de cada ciclo esta tabla está vacía.

Luego, por cada uno de los agentes se hace una búsqueda con A*, tomando $\mathcal{T}$ para prohibir posiciones en el espacio y tiempo que han sido ocupadas por otros agentes anteriormente en el ciclo. Una vez encontrado el camino se agregan a $\mathcal{T}$ las primeras `W` posiciones encontradas.

Por otro lado, a diferencia de los algoritmos offline antes mostrados, las búsquedas individuales no utilizan el precómputo clásico con BFS (Sección 2.2.1). Si no que realizan una búsqueda paralela al algoritmo con A* desde cada uno de los objetivos, donde la búsqueda es detenida y reanudada entre fases dependiendo de las heurísticas que vayan siendo necesarias por el algoritmo principal. Todo esto gracias a la naturaleza de los algoritmos online, que dan la oportunidad de recalcular y reutilizar información entre las distintas fases de planeamiento.

Una vez terminada esta fase, se pasa a la fase de ejecución, moviendo a todos los agentes $\mathtt{W}/2$ pasos. Luego se repite el ciclo, vaciando la tabla $\mathcal{T}$ y partiendo las búsquedas desde las nuevas posiciones de los agentes.

Es importante mencionar que el algoritmo es muy sensible al orden de resolución de los agentes. Es fácil pensar en ejemplos en donde seguir cierto orden entre dos agentes producirá una solución de cierto costo, pero donde el orden reverso podría producir una solución de mayor costo, o inclusive no permitir al algoritmo encontrar una solución. Es por esto, que, en cada iteración del algoritmo, el orden de resolución de los agentes es cambiado. Este orden es arbitrario, pero en la implementación original se propone el uso de un sistema *round robin*.

**Algoritmo 5**: WHCA*

**Input**: Un problema MAPF $(G = (V, E), \mathcal{A}, init, goal)$, W

1 **while** *Algún agente no está en su objetivo* **do**
2 $\quad \mathcal{T} \leftarrow \emptyset$
3 $\quad ord(\mathcal{A}) \leftarrow$ reordenar $\mathcal{A}$
4 $\quad sol_{parcial} \leftarrow \emptyset$
5 $\quad$ **forall** $a \in ord(\mathcal{A})$ **do**
6 $\quad\quad \pi \leftarrow encontrarCamino(a, \mathcal{T})$
7 $\quad\quad$ Reservar los primeros W pasos de $\pi$ en $\mathcal{T}$
8 $\quad\quad sol_{parcial}[a_i] \leftarrow \pi$
9 $\quad$ **end**
10 $\quad$ Mover en paralelo $\mathtt{W}/2$ pasos los agentes según los caminos de $sol_{parcial}$
11 **end**

## 5.4. ASP con ventanas

Como se vió en la sección 4, los resultados de nuestro solver offline muestran un excelente rendimiento en grillas compactas donde existe una gran cantidad de conflictos posibles.

Motivados por estos resultados y utilizando la idea del uso de ventanas en algoritmos como WHCA*, proponemos un nuevo solver online basado en nuestra codificación en ASP que resuelve MAPF bajo la función de suma de costos. En esta versión se resuelve en incrementos de W instantes de tiempo, solo considerando los conflictos entre agentes para esos primeros W pasos. Este número es el equivalente al makespan de nuestro solver offline, y al igual que en la codificación anterior es entregado como parámetro.

A continuación, mostramos la nueva codificación propuesta que usa nuestro solver.

### 5.4.1. Nuevos Átomos

Para nuestra nueva codificación definimos los siguientes átomos además de los ya existentes en la sección 3.1.1:

- $in_range(a)$: indica que $a$ es un agente que tiene su objetivo dentro del rango de la ventana. Esto es útil para redefinir el logro de objetivos.
- $exit_penalty(a, c)$: indica el costo $c$ de salir de una celda objetivo para un agente $a$. A diferencia de nuestra codificación online, al salir de una celda un agente podría tener que pagar un costo mayor a 1. Explicamos los detalles más adelante, en la sección 5.4.4.
- $window_time(t)$: $t$ es un instante dentro de la ventana de tiempo, este sobre escribe $time(t)$, cambiamos el nombre simplemente para diferenciar las codificaciones.

### 5.4.2. Codificación de los efectos

Nuestra compilación se mantiene en cuanto a codificación del mundo con respecto a las secciones 3.1 y 3.3. Sin embargo, surge un nuevo problema: Si consideramos que el movimiento de los agentes está descrito por una regla como la mostrada en 3.1, debido a que no existe una celda final para los agentes en la cual deben estar, entonces no hay razón que obligue a nuestro solver a tomar una acción que lleve al agente a una posición ilegal haciendo falso al predicado $delta$ y por lo tanto, no estaría generando nuevos predicados $at(a, x, y, t)$.

Para manejar esto, es necesario indicar a los agentes que deben tener al menos un predicado $at(a, x, y, t)$ verdadero por cada instante $t$.

Por lo que agregamos la siguiente restricción:

$$\leftarrow agent(A), \{at(A, X, Y, T) : rangeX(X), rangeY(Y)\}0,$$
$$window_time(T).$$

Esta restricción, obliga a todos los agentes a estar en una posición en cada instante de tiempo, indicando que por cada instante de tiempo $t$ cada uno de los agentes $a$ no puede tener 0 predicados $at(a, x, y, t)$ verdaderos.

### 5.4.3. Logro del Objetivo

A diferencia de la codificación offline no existe la obligación de llevar los agentes a sus respectivos objetivos, si no que los agentes solo se acercarán hacia ellos como resultado de la minimización de costos. Por lo tanto a diferencia de la sección 3.1.8, solo definimos el predicado $at_goal(a, t)$.

$$at_goal(A, T) \leftarrow at(A, X, Y, T), goal(A, X, Y), \tag{5.1}$$

### 5.4.4. Optimización de costos

Para codificar la función de costos, usamos la misma definición en la sección 3.2.1. Sin embargo, agregamos y modificamos ciertas reglas.

Debido a que ahora la última celda a la que llega un agente no necesariamente es su objetivo. debemos penalizarlo según el costo faltante hasta el objetivo. Dado que el costo no es sabido de antemano, utilizamos el precómputo de la sección 3.4. De esta manera, añadimos la siguiente regla, que agrega el valor de $cost_to_go$ de la última celda alcanzada. [1]

$$\begin{aligned} cost(A, -1, C) \leftarrow & at(A, X, Y, \mathtt{W}), \\ & cost_to_go(A, X, Y, C). \end{aligned}$$

Además, debemos cambiar la definición de $at_goal_back$, para que solo sea considerado si es que el agente llega a su objetivo.

[1] Se indica $T = -1$ para identificar estos tipos de costos y así no sobrescriban los costos indicados por las otras reglas.

$$
\begin{aligned}
at_goal_back(A, \mathtt{W}) &\leftarrow agent(A), at_goal(A, \mathtt{W}), in_range(A), \\
at_goal_back(A, T-1) &\leftarrow at_goal_back(A, T), exec(A, wait, T-1).
\end{aligned}
$$

Dada la siguiente situación: Un agente $a$ se encuentra en su objetivo $goal(a)$ luego de una iteración $i$ del algoritmo, debido a que este ya llegó a su objetivo existen dos posibilidades a futuro:

- El agente se mantiene en $goal(a)$ hasta el término del algoritmo: Esto significa que el agente no incurre en costos desde la iteración $i$ del algoritmo, y por lo tanto no se deberá penalizar desde ese punto en adelante. Esto es controlado con las reglas de la sección 3.2.1, que garantizan que no hayan penalizaciones dentro de una misma ventana.
- El agente ejecuta una acción distinta a $esperar$ en una iteración $j$ posterior. Esto significa que el agente debe ser penalizado según el total de tiempo transcurrido entre iteraciones. Para esto, agregamos la siguiente regla[2]:

$$
\begin{aligned}
cost(A, -2, C) \leftarrow & exec(A, M, T), M \neq esperar, \\
& exit_penalty(A, C).
\end{aligned}
$$

  Donde $exit_penalty(a, c)$ es calculado en cada iteración del algoritmo, e indica justamente el tiempo transcurrido desde la última vez que $a$ había llegado a su objetivo. Esta penalización es tomada en cuenta si es que un agente ejecuta una acción distinta de $esperar$. Cabe destacar que $exit_penalty$ tiene valor 0 si es que el agente no se encuentra en una celda objetivo.

[2]Se indica $T = -2$ por las mismas razones descritas anteriormente.

### 5.4.5. Algoritmo

Este nuevo solver, al igual que WHCA* recibe como input un problema en MAPF $(G = (V, E), \mathcal{A}, init, goal)$, un `W` que indica el tamaño de las ventanas. El pseudocódigo está definido en el Algoritmo 6

El algoritmo primero parte con todos los agentes en sus posiciones originales indicadas por $init$. Definimos las tablas $exit_penalty$, $in_range$ y $cost_to_go$, que servirán para definir los valores de los predicados definidos en las secciones anteriores.

Luego, se crea y resuelve un programa en ASP con la codificación mostrada anteriormente, indicando los valores de las tablas, junto con las posiciones actuales de los agentes. Finalmente, los agentes son movidos `W` pasos, actualizando $init$ según las nuevas posiciones finales indicadas por el programa. Cabe notar que decidimos fijar el movimiento en `W` pasos (en contraste a los `W`$/2$ pasos de WHCA*) porque fue el valor que mejor rendimiento y tasa de éxito tuvo en nuestros experimentos.

Los valores de las tablas son recalculados en cada iteración del algoritmo.

- $in_range$: Al inicio de cada iteración se revisa por cada uno de los agentes si es que el costo óptimo $c^*(init(a), goal(a))$ sea menor al tamaño de la ventana `W`, definiendo $in_range[a] = 1$ si es que está en el rango de la ventana, y $in_range[a] = 0$ en caso contrario.
- $costs_to_go$: Al inicio de cada iteración, se vacía la tabla y se agregan solamente las tuplas $cost_to_go(a, x, y, c)$ si es que la celda $(x, y)$ está a una distancia menor o igual a `W` desde $init(a)$. Donde el valor de $c$ corresponde a $c^*((x, y), goal(a))$.
- $exit_penalty$: Es calculado al final de cada iteración, donde el valor se define dependiendo de los siguientes casos:
  - Si el agente $a$ no termina en su celda objetivo $goal(a)$, entonces ocurre que $exit_penalty[a] = 0$. Es decir, no habrán penalizaciones por salir de su celda.

- Si el agente $a$ termina en su celda objetivo, entonces definimos el valor de $exit_penalty[a] = t_{obj}$, que se define como el tiempo transcurrido desde la última vez en que el agente se movió hacia el $goal(a)$.

Todo este proceso se repite constantemente hasta que todos los agentes lleguen a sus respectivos objetivos.

Es importante mencionar que en cada iteración del algoritmo son calculadas las distancias entre celdas y cada uno de los objetivos. Esto se hace utilizando el método indicado en la sección 2.2.1. Sin embargo, utilizando las ideas de WHCA*, las búsquedas solo se ejecutan hasta encontrar las distancias de las celdas que van a ser utilizadas en la iteración actual, es decir, la búsqueda se detiene una vez que encuentra todas las celdas alcanzables en `W` pasos desde $init(a)$. Estas búsquedas son nuevamente reanudadas en las siguientes iteraciones si es que existen distancias que vayan a ser necesarias en la ejecución de la ventana actual.

**Algoritmo 6**: MAPF en ASP con ventanas

```
Input: Un problema MAPF (G = (V, E), A, init, goal), W
exit_penalty ← Tabla de tamaño |A| inicializada en 0.
in_range ← Tabla de tamaño |A| inicializada en 0.
Partir precómputo de A*.
while Algún agente no está en su objetivo do
    costs_to_go ← Lista vacía
    forall a ∈ A do
        Consultar A* por las distancias c*((x, y), goal(a)), donde (x, y)
        es alcanzable por a en W pasos.
        if c*(init(a), goal(a)) ≤ W then
            in_range[a] ← 1
        else
            in_range[a] ← 0
        end
        Agregar cost_to_go(a, x, y, c) a costs_to_go por todas las celdas (x, y)
        que están a una distancia menor o igual a W desde init(a).
    end
    P(W, exit_penalty, costs_to_go, in_range) ← Crear un programa en ASP
    con el problema y sus nuevas posiciones.
    M ← resolver P.
    Mover en paralelo W pasos los agentes según los caminos de M .
    forall a ∈ A do
        init(a) ← Última posición de a luego de mover agentes.
        if a se encuentra en goal(a) then
            exit_penalty[a_i] ← Recalcular tiempo desde que estuvo por
            última vez en el objetivo
        else
            exit_penalty[a_i] ← 0
        end
    end
end
```

# 6. EXPERIMENTOS CON VENTANAS

Con el objetivo de analizar el rendimiento de nuestro nuevo solver online se realizaron experimentos sobre distintas instancias de MAPF utilizando el solver WHCA* como benchmark. Se probaron distintos tamaños de ventanas sobre un número variable de agentes. A diferencia de los experimentos de nuestra versión offline, la evaluación estuvo enfocada en grillas de mayor tamaño, y por consiguiente problemas con una mayor cantidad de agentes, para así efectivamente enfocarnos en el rendimiento sobre el tipo de problemas que no logra resolver nuestro solver offline.

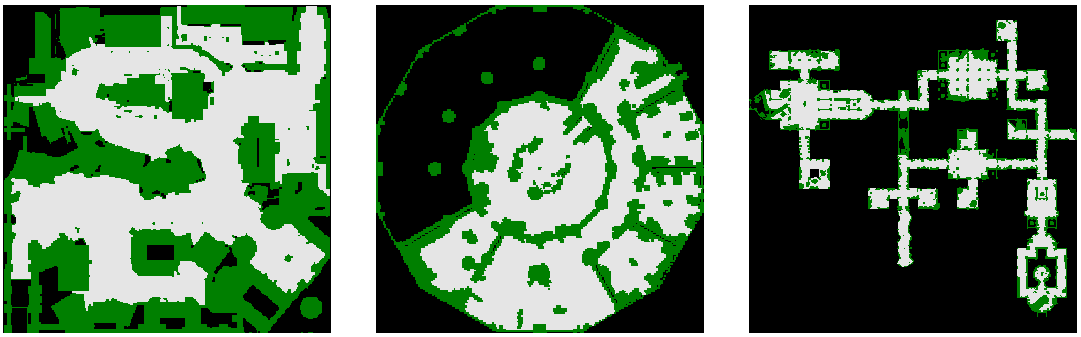
Nuestros experimentos tuvieron especial énfasis en el análisis de como el tamaño de las ventanas afecta el rendimiento y costo de las soluciones para ambos solvers, y como además este afecta el proceso de instanciación. También analizamos como cambia la suboptimalidad de las soluciones según estos factores, permitiéndonos escoger las variables adecuadas para encontrar las configuraciones ideales según los problemas a resolver.

Al igual que todos los experimentos anteriores, el código de nuestra implementación está escrito en Python 3.7, utilizando Clingo 5.3 (Gebser y cols., 2014) como nuestro solver ASP. Clingo ejecutado con 4 threads en su modo paralelo, y USC como estrategia de optimización. WHCA* fue programado de forma propia en `C++`.

Todos los experimentos fueron ejecutado en un computador con procesador Intel Core i5-3570K con 3.40GHz y con 8GB de memoria RAM. Para todas nuestras ejecuciones fijamos un tiempo límite de 5 minutos.

En concreto, experimentamos en mapas estándar basados en el juego Dragon Age: Origins (Sturtevant, 2012). Por cada uno de ellos generamos $10$ instancias con distintas posiciones para los agentes en $\{25, 50 \ldots, 300\}$, , entregando un total de 120 problemas distintos. Nuestra implementación fue ejecutada con $\mathtt{W} \in \{4, 8, 12, 16\}$. WHCA* fue ejecutado con $\mathtt{W} \in \{4, 8, 16, 32\}$.

Específicamente se probaron en los siguientes mapas: (mostrados en la Figura 6.1)



(a) den520d (b) ost003d (c) brc202d

Figura 6.1. Mapas DAO

## 6.1. Mapa den520d

Primero, experimentamos en el mapa den520d (Figura 6.1a), el cual posee muchos espacios abiertos, pero con varios puntos posibles de conflictos gracias a la existencia de pasillos pequeños en secciones importantes. El tamaño de este mapa es de $256 \times 257$.

Observando los resultados de tasa de éxito según el número de agentes, y el número de problemas resueltos en el tiempo (mostrados en las Figuras 6.2 y 6.3) respectivamente, podemos apreciar un claro mejor desempeño de WHCA* para todos los tamaños de las ventanas. Donde el único tamaño para ASP que presenta un rendimiento comparable es 4.

También reportamos los costos promedio y tiempo promedio de resolución encontrados en las Tablas 6.1 y 6.2 respectivamente. Solo reportamos $\mathtt{W} \in \{4, 8\}$ para ASP y $\mathtt{W} \in \{4, 8, 16\}$ para WCHA* debido a que estas son las versiones con mayor tasa de éxito. Los valores promedio solo consideran las instancias resueltas en común por cada uno de los algoritmos.

No es sorprendente ver que un mayor valor para el tamaño de las ventanas entrega menores costos y mayores tiempos de resolución exceptuando el caso de ASP con $25$

agentes, lo cual se explica debido a que la menor cantidad de agentes significa una menor probabilidad de conflicto entre ellos, haciendo que sea menos costoso resolver estas instancias.

Si bien los resultados en general muestran un mejor rendimiento para WHCA*, encontramos soluciones de costo mucho menor utilizando nuestra codificación, donde la versión de menor tiempo de resolución, ASP(4), tiene resultados de costos mucho más cercanos al valor óptimo que todas las versiones de WHCA* probadas.

La explicación de esto se debe a la implementación de penalties de salida de objetivos indicada en la sección 5.4.4, donde a diferencia de WHCA*, nuestro solver optimiza teniendo en cuenta estos costos, intentando no mover a los agentes que ya se encuentran en sus respectivos objetivos.

Estos últimos puntos los podemos ver reflejados con mayor claridad en la Tabla 6.3. En esta podemos ver los costos promedio de los problemas que alcanzaron a ser resueltos por ICBS-h, que justamente corresponden al valor óptimo. Además, ejecutamos una versión de nuestra codificación de ASP con $\mathtt{W} = 4$ que ignora los costos de salida de objetivos en su optimización, indicada como ASP(4)$^{-}$. Esto con el objetivo de generar una codificación con una optimización más parecida a WHCA*, para así analizar posibles ventajas y desventajas en estos enfoques. También, por cada uno de estos algoritmos agregamos la columna *Subopt*, que indica la suboptimalidad de los costos de la solución. Observando estos resultados es evidente la importancia de la codificación de estos penalties, donde tanto como ASP$^{-}$ y WHCA* muestran una suboptimalidad substancialmente mayor a ASP. Por otro lado no reportamos mayores cambios en cuanto al tiempo de ejecución de ASP(4)$^{-}$ y ASP(4).

Por último, en la Figura 6.4 se ilustra el tiempo ocupado en el proceso de instanciación versus el tiempo total de ejecución promedio de las instancias resueltas por ASP(4) y

ASP(8). Además en la Tabla 6.4 reportamos los tiempos promedio de los primeros problemas y los tiempos promedio para el máximo número de agentes en el cual el solver aún encuentra al menos una solución. Todo esto por cada uno de los valores de $\mathtt{W} \in \{4, 8, 12, 16\}$. Donde, $Grd_0$ indica el promedio en segundos ocupado en instanciar los problemas con la menor cantidad de agentes, es decir, los problemas de $25$ agentes. $Total_0$ corresponde al tiempo total promedio de resolución para las primeras instancias. $Porc_0$ indica el porcentaje promedio del tiempo total utilizado en el proceso de instanciación. $Grd_f$, $Total_f$, y $Porc_f$ se definen igual que los valores anteriores, pero para los problemas con el número máximo de agentes resolvible por el solver.

Observamos que para valores menores de `W` el proceso de instanciación toma mayor importancia en términos porcentuales, lo cual tiene sentido debido a que para ventanas de mayor tamaño la dificultad de resolver los subproblemas generados por los límites de las ventanas es mayor. Esta tendencia se repite al aumentar el número de agentes, donde el aumento de estos hace que aumente el tiempo de resolución de los modelos en ASP.

También observamos una tendencia a la estabilización en el tiempo de instanciación con mayores valores de `W`, y un valor bastante elevado en general para todas las instancias. Esto indica un problema importante en el proceso de instanciación, dado el hecho de que si bien la resolución de los problemas está separada en subsecciones gracias a estas ventanas, la instanciación no está completamente separada, donde varias reglas/clausulas aún tienen que ser expandidas/generadas por cada una de las celdas del mapa.

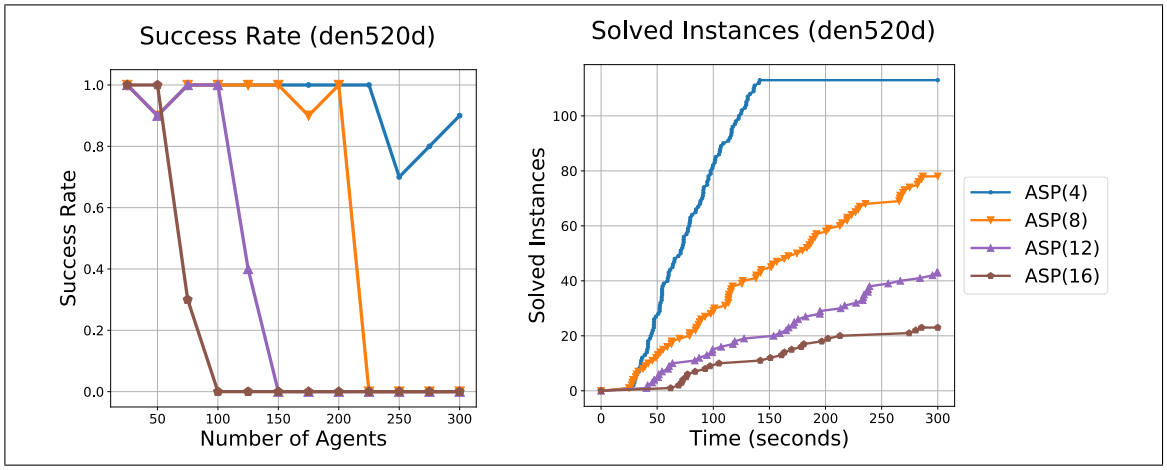



Figura 6.2. Tasa de éxito y número de instancias resueltas en el tiempo en mapa den520d utilizando codificación ASP. Apreciamos un mejor rendimiento cuando W es disminuido.

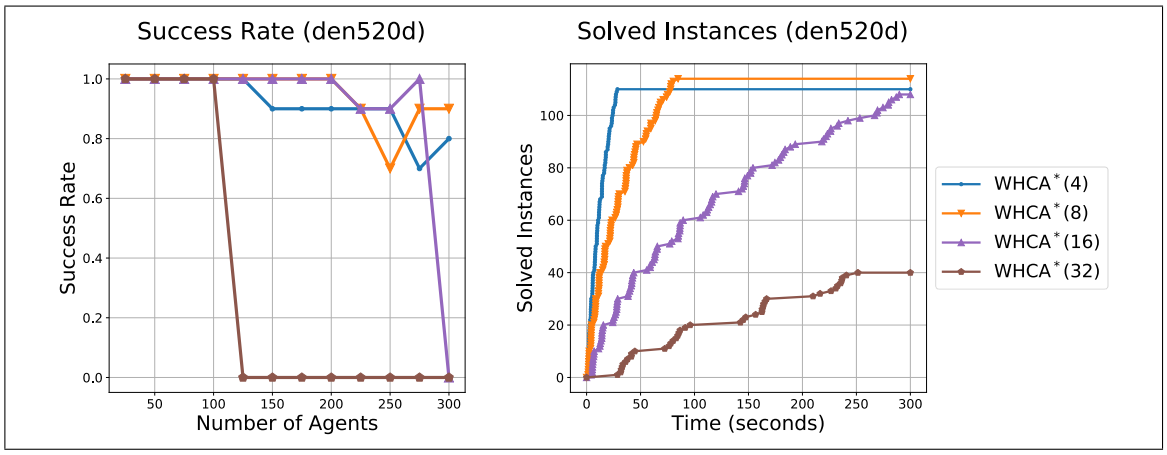



Figura 6.3. Tasa de éxito y número de instancias resueltas en el tiempo en mapa den520d utilizando WHCA*. Apreciamos un mejor rendimiento cuando W es disminuido. WHCA* además muestra mejores resultados comparado con ASP.

Tabla 6.1. Costo promedio de las instancias resueltas en común utilizando distintos solvers y configuraciones sobre mapa den520d, aumentando el número de agentes. Se aprecian mejores resultados al aumentar el tamaño de las ventanas. Por otro lado, ASP entrega soluciones más óptimas en todos los casos.

| **# agentes** | **resueltas** | **Costo de Solución promedio** | | | | |
|---|---|---|---|---|---|---|
| | | **ASP(4)** | **ASP(8)** | **WHCA*(4)** | **WHCA*(8)** | **WHCA*(16)** |
| 25 | 10 | 3411,5 | 3412,1 | 3457,0 | 3460,9 | 3477,3 |
| 50 | 9 | 6480,3 | 6479,6 | 6595,7 | 6588,2 | 6606,3 |
| 75 | 10 | 10531,5 | 10527,2 | 10763,7 | 10776,3 | 10767,4 |
| 100 | 10 | 13771,9 | 13758,0 | 14316,5 | 14252,0 | 14167,9 |
| 125 | 10 | 17321,2 | 17303,1 | 17997,9 | 18041,0 | 17928,3 |
| 150 | 9 | 20921,4 | 20900,4 | 22109,2 | 22033,2 | 22138,7 |
| 175 | 9 | 24166,0 | 24127,3 | 25574,3 | 25373,1 | 25385,7 |
| 200 | 9 | 28079,0 | 28026,9 | 29761,1 | 29658,3 | 29829,3 |
| 225 | 0 | −1,0 | −1,0 | −1,0 | −1,0 | −1,0 |

Tabla 6.2. Tiempo de solución promedio de instancias resueltas en común utilizando distintos solvers y configuraciones sobre mapa den520d, aumentando el número de agentes. Se aprecian mejores resultados al disminuir el tamaño de las ventanas. WHCA* tiene mejor rendimiento, y los resultados son similares entre ASP(4) y WHCA*(16).

| **# agentes** | **resueltas** | **Tiempo de Solución promedio (s)** | | | | |
|---|---|---|---|---|---|---|
| | | **ASP(4)** | **ASP(8)** | **WHCA*(4)** | **WHCA*(8)** | **WHCA*(16)** |
| 25 | 10 | 34,9 | 32,3 | 1,2 | 2,0 | 5,8 |
| 50 | 9 | 37,4 | 56,4 | 2,4 | 4,2 | 13,7 |
| 75 | 10 | 47,8 | 88,1 | 3,9 | 7,9 | 26,9 |
| 100 | 10 | 54,0 | 114,0 | 5,6 | 11,5 | 41,1 |
| 125 | 10 | 62,9 | 148,6 | 7,8 | 16,5 | 61,8 |
| 150 | 9 | 73,5 | 187,8 | 9,8 | 22,3 | 85,5 |
| 175 | 9 | 81,9 | 223,5 | 11,7 | 28,6 | 112,5 |
| 200 | 9 | 93,0 | 274,1 | 14,5 | 36,6 | 146,9 |
| 225 | 0 | −1,0 | −1,0 | −1,0 | −1,0 | −1,0 |

Tabla 6.3. Costo de solución promedio de instancias resueltas en común utilizando distintos solvers y configuraciones sobre mapa den520d, aumentando el número de agentes. Donde *Subopt* corresponde a la suboptimalidad de los solvers con respecto a ICBS-h. ICBS-h entrega el costo de la solución óptima. ASP$^-$ corresponde a una codificación que ignora los costos de salida de objetivos. Apreciamos que los costos de las soluciones son parecidos entre ASP(4)$^-$ y WHCA*(16).

| **# agentes** | **resueltas** | **Costo de Solución promedio** | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **ICBS-h** | **ASP(4)** | ***Subopt*** | **ASP(4)$^-$** | ***Subopt*** | **WHCA*(16)** | ***Subopt*** |
| 25 | 10 | 3409,7 | 3411,5 | 1,001 | 3413,4 | 1,001 | 3477,3 | 1,020 |
| 50 | 9 | 6472,7 | 6480,3 | 1,001 | 6537,8 | 1,010 | 6606,3 | 1,021 |
| 75 | 9 | 10521,6 | 10540,8 | 1,002 | 10687,0 | 1,016 | 10772,4 | 1,024 |
| 100 | 7 | 13501,6 | 13540,4 | 1,003 | 13812,4 | 1,023 | 13959,3 | 1,034 |
| 125 | 6 | 17210,5 | 17284,5 | 1,004 | 18113,5 | 1,052 | 17931,5 | 1,042 |
| 150 | 2 | 21580,0 | 21660,0 | 1,004 | 22110,0 | 1,025 | 22919,5 | 1,062 |
| 175 | 1 | 23101,0 | 23176,0 | 1,003 | 23900,0 | 1,035 | 23894,0 | 1,034 |
| 200 | 0 | −1,0 | −1,0 | 1,000 | −1,0 | 1,000 | −1,0 | 1,000 |

Tabla 6.4. Tiempos de instanciación promedio en mapa den520d con distintos valores de `W`. *Grd* indica el promedio en segundos ocupado en instanciar los problemas. *Total* corresponde al tiempo total promedio de resolución. *Porc* indica el porcentaje promedio del tiempo total utilizado en instanciación. Notamos que el proceso de instanciación toma menor importancia al aumentar el número de agentes.

| **ASP** | **Max. agentes** | ***Grd*$_0$ (s)** | ***Total*$_0$ (s)** | ***Porc*$_0$** | ***Grd*$_f$ (s)** | ***Total*$_f$ (s)** | ***Porc*$_f$** |
|---|---|---|---|---|---|---|---|
| $\mathtt{W} = 4$ | 300 | 33,4 | 34,9 | 95,6 % | 92,1 | 133,4 | 69,1 % |
| $\mathtt{W} = 8$ | 200 | 25,9 | 32,3 | 80,2 % | 145,5 | 275,0 | 52,9 % |
| $\mathtt{W} = 12$ | 125 | 36,6 | 51,5 | 71,1 % | 153,4 | 286,3 | 53,6 % |
| $\mathtt{W} = 16$ | 75 | 53,9 | 80,8 | 66,7 % | 158,6 | 280,2 | 56,6 % |

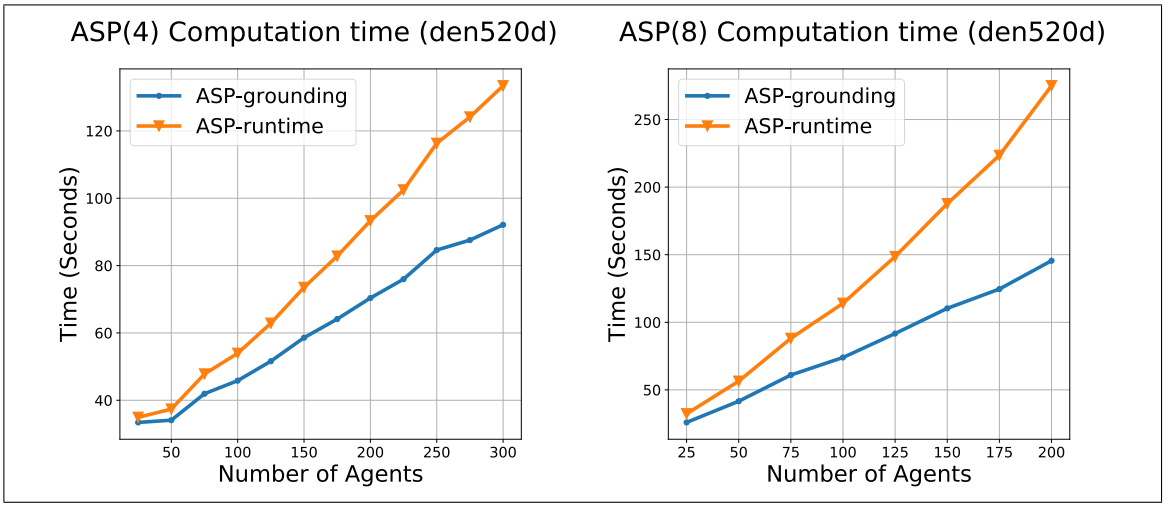



Figura 6.4. Tiempo total de ejecución e instanciación requerido en mapa den520d a medida que aumenta el número de agentes, utilizando ASP con $\mathtt{W} = 4$ y $\mathtt{W} = 8$. Vemos que el proceso de instanciación toma tiempo importante en ambos casos, con una tendencia a disminuir comparativamente al aumentar el tamaño de las ventanas.

## 6.2. Mapa ost003d

Luego, experimentamos en el mapa ost003d (Figura 6.1b), que al igual que el mapa anterior, posee muchos espacios abiertos, sin embargo, no tiene pasillos importantes y por lo tanto tiene menos puntos de conflictos probables. El tamaño de este mapa es de $194 \times 194$.

Los resultados de tasa de éxito y número de problemas resueltos en el tiempo están indicados en las Figuras 6.5 y 6.6. Los costos promedio y tiempo promedio de resolución se encuentran en las Tablas 6.5 y 6.6. Los tiempos de computo totales y de instanciación se muestran en la Figura 6.7 y en la Tabla 6.8.

Observando los resultados, podemos ver que se repite el comportamiento del mapa anterior, con un mejor desempeño de WHCA* en general y costos más cercanos al óptimo por parte de ASP.

Aun así, cabe destacar que el desempeño de WHCA* es aún mejor que en los resultados anteriores, donde por ejemplo el tiempo promedio de las últimas instancias resueltas por ASP(4) es $82{,}4$ y para WHCA*(16) es $99{,}7$, en cambio en den520d son $93{,}0$ y $146{,}9$ respectivamente. Tomando en cuenta el hecho de que ambos mapas tienen un tamaño bastante similar, es que estos resultados reafirman el mejor rendimiento en grillas con mayor congestión para ASP.

Por otro lado, no observamos una gran diferencia de rendimiento para ASP en los dos mapas al aumentar el tamaño de `W` al observar las Figuras 6.5 y 6.6. También observamos un comportamiento bastante similar en los tiempos de instanciación, con un leve mejor rendimiento para ost003d.

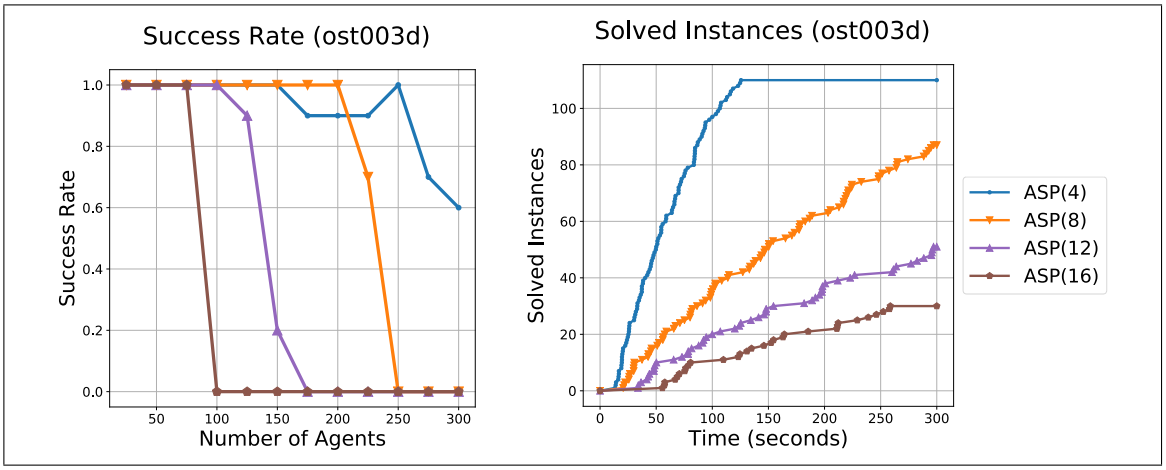



Figura 6.5. Tasa de éxito y número de instancias resueltas en el tiempo en mapa ost003d utilizando codificación ASP. Apreciamos un mejor rendimiento cuando W es disminuido.

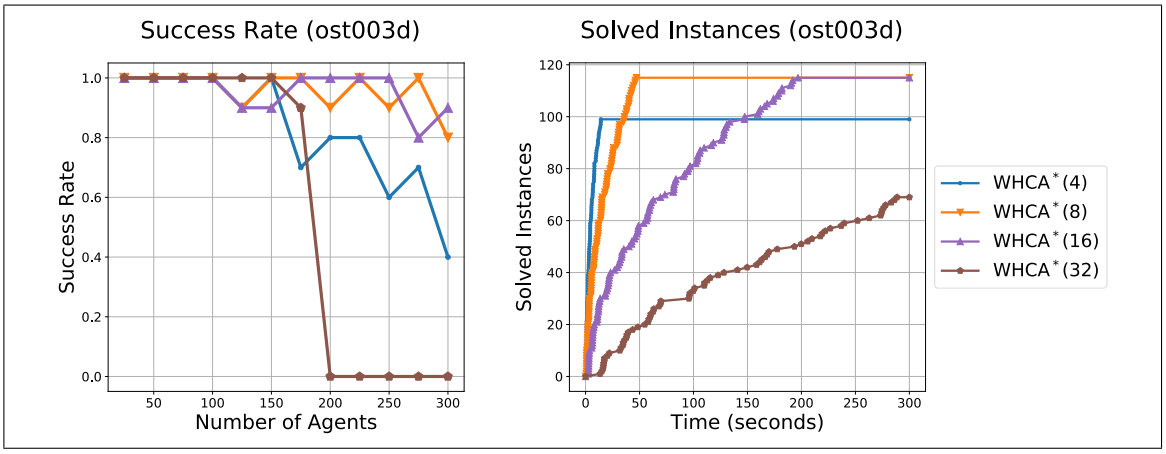



Figura 6.6. Tasa de éxito y número de instancias resueltas en el tiempo en mapa ost003d utilizando WHCA*. Apreciamos un mejor rendimiento cuando W es disminuido. WHCA* además muestra mejores resultados comparado con ASP.

Tabla 6.5. Costo promedio de las instancias resueltas en común utilizando distintos solvers y configuraciones sobre mapa ost003d, aumentando el número de agentes. Se aprecian mejores resultados al aumentar el tamaño de las ventanas. Por otro lado, ASP entrega soluciones más óptimas en todos los casos.

| **# agentes** | **resueltas** | **Costo de Solución promedio** | | | | |
|---|---|---|---|---|---|---|
| | | **ASP(4)** | **ASP(8)** | **WHCA*(4)** | **WHCA*(8)** | **WHCA*(16)** |
| 25 | 10 | 2893,6 | 2893,3 | 2989,0 | 2994,7 | 2972,7 |
| 50 | 10 | 5295,8 | 5293,8 | 5476,3 | 5496,3 | 5432,9 |
| 75 | 10 | 7488,7 | 7480,3 | 8035,2 | 8171,5 | 8024,0 |
| 100 | 10 | 10586,3 | 10566,8 | 11379,8 | 11341,9 | 11243,2 |
| 125 | 8 | 13706,4 | 13685,2 | 14711,1 | 14717,6 | 14777,8 |
| 150 | 9 | 16715,7 | 16678,1 | 18220,0 | 18133,8 | 18012,9 |
| 175 | 6 | 18846,0 | 18803,7 | 20581,7 | 20793,8 | 20785,5 |
| 200 | 6 | 22422,3 | 22358,3 | 24892,8 | 24730,2 | 24828,3 |
| 225 | 6 | 23889,7 | 23817,0 | 26539,7 | 26584,5 | 26695,0 |
| 250 | 0 | −1,0 | −1,0 | −1,0 | −1,0 | −1,0 |

Tabla 6.6. Tiempo de solución promedio de instancias resueltas en común utilizando distintos solvers y configuraciones sobre mapa ost003d, aumentando el número de agentes. Se aprecian mejores resultados al disminuir el tamaño de las ventanas. WHCA* tiene mejor rendimiento, y los resultados son similares entre ASP(4) y WHCA*(16).

| **# agentes** | **resueltas** | **Tiempo de Solución promedio (s)** | | | | |
|---|---|---|---|---|---|---|
| | | **ASP(4)** | **ASP(8)** | **WHCA*(4)** | **WHCA*(8)** | **WHCA*(16)** |
| 25 | 10 | 16,8 | 26,3 | 0,4 | 0,9 | 3,0 |
| 50 | 10 | 22,0 | 48,7 | 0,8 | 1,9 | 6,7 |
| 75 | 10 | 29,1 | 74,9 | 1,4 | 3,2 | 11,8 |
| 100 | 10 | 37,2 | 101,1 | 2,2 | 5,3 | 20,7 |
| 125 | 8 | 46,1 | 135,5 | 3,1 | 8,1 | 31,6 |
| 150 | 9 | 53,4 | 170,5 | 4,1 | 11,3 | 45,6 |
| 175 | 6 | 65,4 | 212,7 | 5,2 | 14,9 | 59,4 |
| 200 | 6 | 71,5 | 243,9 | 6,5 | 19,4 | 83,6 |
| 225 | 6 | 82,4 | 288,1 | 7,9 | 24,1 | 99,7 |
| 250 | 0 | −1,0 | −1,0 | −1,0 | −1,0 | −1,0 |

Tabla 6.7. Costo de solución promedio de instancias resueltas en común utilizando distintos solvers y configuraciones sobre mapa ost003d, aumentando el número de agentes. Donde *Subopt* corresponde a la suboptimalidad de los solvers con respecto a ICBS-h. Vemos resultados más cercanos al óptimo por parte de ASP.

| **# agentes** | **resueltas** | **Costo de Solución promedio** | | | | |
|---|---|---|---|---|---|---|
| | | **ICBS-h** | **ASP(4)** | ***Subopt*** | **WHCA*(16)** | ***Subopt*** |
| 25 | 10 | 2890,8 | 2893,6 | 1,001 | 2972,7 | 1,028 |
| 50 | 10 | 5287,6 | 5295,8 | 1,002 | 5432,9 | 1,027 |
| 75 | 9 | 7493,9 | 7519,8 | 1,003 | 8075,4 | 1,078 |
| 100 | 0 | −1,0 | −1,0 | 1,000 | −1,0 | 1,000 |
| 125 | 1 | 13256,0 | 13291,0 | 1,003 | 13910,0 | 1,049 |
| 150 | 0 | −1,0 | −1,0 | 1,000 | −1,0 | 1,000 |

Tabla 6.8. Tiempos de instanciación promedio en mapa ost003d con distintos valores de `W`. *Grd* indica el promedio en segundos ocupado en instanciar los problemas. *Total* corresponde al tiempo total promedio de resolución. *Porc* indica el porcentaje promedio del tiempo total utilizado en instanciación. Notamos que el proceso de instanciación toma menor importancia al aumentar el número de agentes.

| **ASP** | **Max. agentes** | ***Grd*$_0$ (s)** | ***Total*$_0$ (s)** | ***Porc*$_0$** | ***Grd*$_f$ (s)** | ***Total*$_f$ (s)** | ***Porc*$_f$** |
|---|---|---|---|---|---|---|---|
| $\mathtt{W} = 4$ | 300 | 15,4 | 16,8 | 91,9 % | 72,7 | 120,7 | 60,2 % |
| $\mathtt{W} = 8$ | 225 | 20,0 | 26,3 | 75,9 % | 128,7 | 284,6 | 45,2 % |
| $\mathtt{W} = 12$ | 150 | 29,1 | 42,8 | 68,1 % | 131,7 | 296,6 | 44,4 % |
| $\mathtt{W} = 16$ | 75 | 44,2 | 68,8 | 64,3 % | 120,9 | 230,3 | 52,5 % |

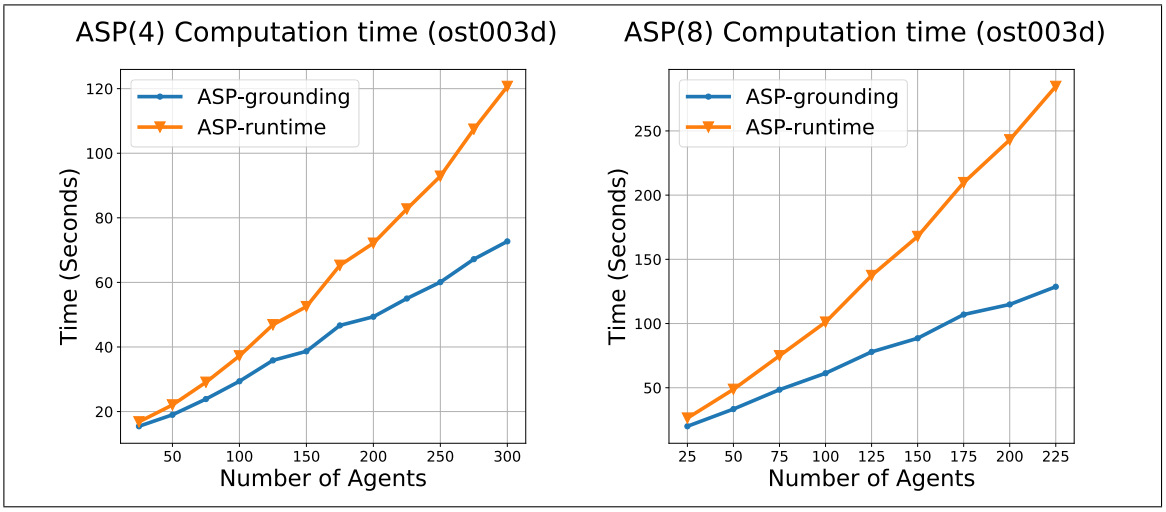



Figura 6.7. Tiempo total de ejecución e instanciación requerido en mapa ost003d a medida que aumenta el número de agentes, utilizando ASP con $\mathtt{W} = 4$ y $\mathtt{W} = 8$. Vemos que el proceso de instanciación toma tiempo importante en ambos casos, con una tendencia a disminuir comparativamente al aumentar el tamaño de las ventanas.

## 6.3. Mapa brc202d

Finalmente, experimentamos en el mapa brc202d (Figura 6.1c), donde, a diferencia de los mapas anteriores, posee muy pocos espacios abiertos y muchos más puntos de posibles conflictos, además tiene un tamaño sustancialmente mayor, siendo este de $530 \times 481$.

Los resultados de tasa de éxito y número de problemas resueltos en el tiempo están indicados en las Figuras 6.8 y 6.9. Los costos promedio y tiempo promedio de resolución se encuentran en las Tablas 6.9 y 6.10. Los tiempos de cómputo totales y de instanciación se muestran en la Figura 6.10 y en la Tabla 6.11.

Nuevamente se repiten los resultados y conclusiones anteriores, con desempeños mejores para WHCA* y costos de solución más pequeños para ASP.

Sin embargo, notamos un peor rendimiento de ASP comparado con los mapas anteriores, donde el tiempo promedio de las últimas instancias resueltas para ASP(4) es $147{,}7$ y para WHCA*(16) es $108{,}3$, siendo el único mapa en que observamos un mejor rendimiento de WHCA*(16). Esto es inesperado debido a la naturaleza del mapa, donde esperamos un mejor rendimiento en instancias como estas, donde existe una mayor congestión y posibles conflictos.

Observando la Figura 6.8 y comparándola con las Figuras 6.2 y 6.5, apreciamos una diferencia importante de los rendimientos al aumentar `W`, donde los dos primeros mapas se ven afectados de mayor manera que en brc202d. Además, viendo los resultados de la Figura 6.10 y la Tabla 6.11, apreciamos un claro aumento del tiempo de instanciación para este mapa en particular comparado con el resto. Todo esto sugiere que el mayor tamaño de este mapa hace que el mejor rendimiento de ASP en mapas congestionados sea opacado por el tiempo dedicado a instanciar el mapa en sí, lo cual explica los resultados inesperados anteriores.

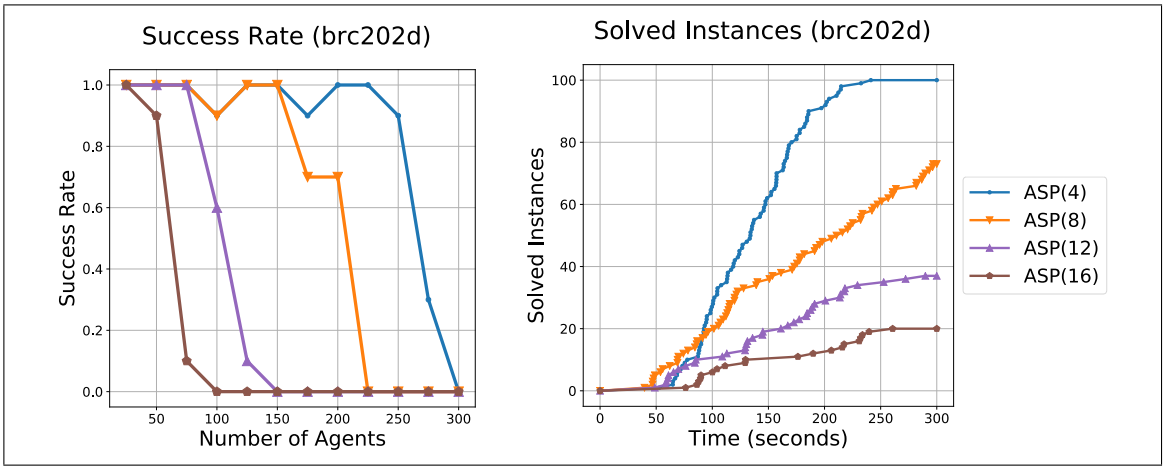



Figura 6.8. Tasa de éxito y número de instancias resueltas en el tiempo en mapa brc202d utilizando codificación ASP. Apreciamos un mejor rendimiento cuando W es disminuido.

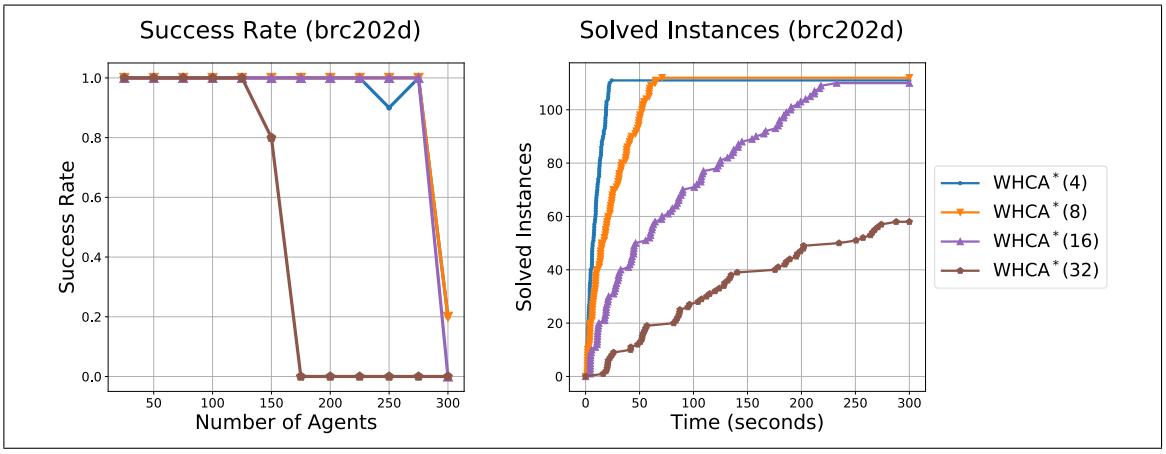



Figura 6.9. Tasa de éxito y número de instancias resueltas en el tiempo en mapa brc202d utilizando WHCA*. Apreciamos un mejor rendimiento cuando W es disminuido. WHCA* además muestra mejores resultados comparado con ASP.

Tabla 6.9. Costo promedio de las instancias resueltas en común utilizando distintos solvers y configuraciones sobre mapa brc202d, aumentando el número de agentes. Se aprecian mejores resultados al aumentar el tamaño de las ventanas. Por otro lado, ASP entrega soluciones más óptimas en todos los casos.

| **# agentes** | **resueltas** | **Costo de Solución promedio** | | | | |
|---|---|---|---|---|---|---|
| | | **ASP(4)** | **ASP(8)** | **WHCA*(4)** | **WHCA*(8)** | **WHCA*(16)** |
| 25 | 10 | 3314,0 | 3314,6 | 3343,1 | 3381,1 | 3402,8 |
| 50 | 10 | 6679,9 | 6673,1 | 6873,2 | 6890,0 | 6910,8 |
| 75 | 10 | 9892,8 | 9885,3 | 10190,4 | 10126,8 | 10196,5 |
| 100 | 9 | 12375,6 | 12368,6 | 12762,2 | 12946,1 | 12935,4 |
| 125 | 10 | 15403,4 | 15386,7 | 16103,7 | 16041,6 | 15963,9 |
| 150 | 10 | 19297,6 | 19276,1 | 20363,0 | 20127,7 | 20227,1 |
| 175 | 7 | 22271,6 | 22242,1 | 23598,4 | 23341,7 | 23446,6 |
| 200 | 7 | 25233,0 | 25194,0 | 26742,1 | 26506,4 | 26625,7 |
| 225 | 0 | −1,0 | −1,0 | −1,0 | −1,0 | −1,0 |

Tabla 6.10. Tiempo de solución promedio de instancias resueltas en común utilizando distintos solvers y configuraciones sobre mapa brc202d, aumentando el número de agentes. Se aprecian mejores resultados al disminuir el tamaño de las ventanas. WHCA* tiene mejor rendimiento, y los resultados son siempre mejores independiente del tamaño de las ventanas.

| **# agentes** | **resueltas** | **Tiempo de Solución promedio (s)** | | | | |
|---|---|---|---|---|---|---|
| | | **ASP(4)** | **ASP(8)** | **WHCA*(4)** | **WHCA*(8)** | **WHCA*(16)** |
| 25 | 10 | 74,4 | 54,0 | 1,1 | 1,8 | 4,6 |
| 50 | 10 | 98,1 | 91,8 | 2,0 | 3,8 | 11,0 |
| 75 | 10 | 97,4 | 111,0 | 3,2 | 6,6 | 19,3 |
| 100 | 9 | 113,8 | 150,9 | 4,2 | 9,3 | 29,4 |
| 125 | 10 | 122,9 | 182,9 | 6,1 | 13,9 | 43,3 |
| 150 | 10 | 142,1 | 231,2 | 8,1 | 19,1 | 62,6 |
| 175 | 7 | 142,6 | 253,9 | 9,7 | 23,7 | 83,2 |
| 200 | 7 | 147,7 | 283,0 | 11,9 | 30,8 | 108,3 |
| 225 | 0 | −1,0 | −1,0 | −1,0 | −1,0 | −1,0 |

Tabla 6.11. Tiempo de instanciación promedio en mapa brc202d con distintos valores de W. *Grd* indica el promedio en segundos ocupado en instanciar los problemas. *Total* corresponde al tiempo total promedio de resolución. *Porc* indica el porcentaje promedio del tiempo total utilizado en instanciación. Notamos que el proceso de instanciación toma menor importancia al aumentar el número de agentes.

| **ASP** | **Max. agentes** | ***Grd*$_0$ (s)** | ***Total*$_0$ (s)** | ***Porc*$_0$** | ***Grd*$_f$ (s)** | ***Total*$_f$ (s)** | ***Porc*$_f$** |
|---|---|---|---|---|---|---|---|
| $\mathtt{W} = 4$ | 275 | 72,3 | 74,4 | 97,2 % | 176,6 | 218,9 | 80,7 % |
| $\mathtt{W} = 8$ | 200 | 47,4 | 54,0 | 87,8 % | 175,4 | 283,0 | 62,0 % |
| $\mathtt{W} = 12$ | 125 | 52,0 | 66,9 | 77,8 % | 174,4 | 289,8 | 60,2 % |
| $\mathtt{W} = 16$ | 75 | 72,2 | 100,5 | 71,8 % | 148,0 | 236,8 | 62,5 % |

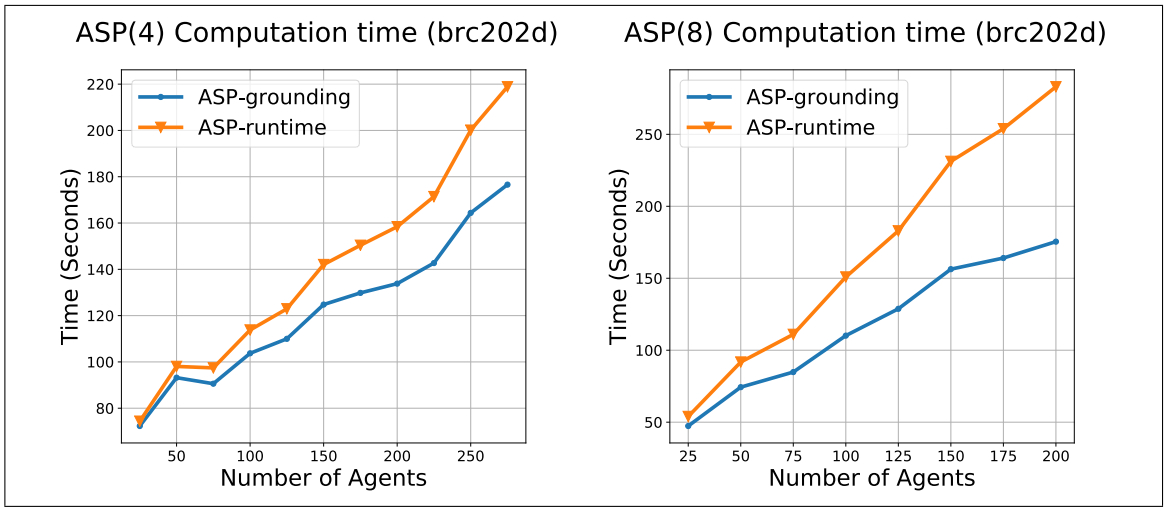



Figura 6.10. Tiempo total de ejecución e instanciación requerido en mapa brc202d a medida que aumenta el número de agentes, utilizando ASP con $\mathtt{W} = 4$ y $\mathtt{W} = 8$. Vemos que el proceso de instanciación toma tiempo importante en ambos casos, con una tendencia a disminuir comparativamente al aumentar el tamaño de las ventanas.

# 7. CONCLUSIONES

En esta tesis propusimos la primera compilación de MAPF bajo la función de suma-de-costos en ASP. Propusimos distintas variantes de este enfoque. En su forma más básica la codificación es cuadrática en el número de agentes, al igual que los enfoques existentes en ASP (por ejemplo Erdem y cols. (2013); Gebser y cols. (2014)) y la codificación de MDD-SAT (Surynek y cols., 2016), un solver en el estado del arte basado en SAT. También propusimos una codificación que es lineal en el número de agentes, y mostramos como mejorar esto ejecutando el algoritmo de BFS o Dijkstra en tiempo de preproceso para generar una codificación más compacta.

En nuestra evaluación empírica en grillas cuadradas, observamos que la codificación lineal supera sustancialmente las codificaciones cuadráticas. En general nuestro enfoque supera en rendimiento los solvers basados en búsqueda y SAT del estado del arte con los cual nos comparamos a medida que aumenta el número de agentes en grillas pequeñas. Esto sugiere que ASP es un enfoque competitivo para resolver instancias de MAPF con alta congestión.

En la segunda parte de nuestra evaluación empírica, estudiamos el rendimiento de nuestro enfoque para encontrar soluciones makespan-óptima, es decir, soluciones en las que, lexicográficamente, el makespan se minimiza primero y el costo se minimiza después. Observamos que este enfoque supera sustancialmente al enfoque original en cuanto a tiempo de resolución y tasa de éxito, mientras que sólo sacrifica ligeramente la optimización; de hecho, las soluciones makespan-óptimas suelen ser menos de un 1 % subóptimas. Esto sugiere que, desde una perspectiva práctica, en las aplicaciones en las que hay una gran congestión, las soluciones makespan-óptimas pueden ser preferibles a las soluciones costo-óptimas.

Nuestro enfoque, al igual que otros enfoques basados en compilaciones, no escala a grillas grandes. De hecho, en esas grillas el tiempo de instanciación se hace extremadamente grande, haciendo nuestro enfoque impráctico. Una línea abierta de investigación es

incorporar nuestras ideas a enfoques distribuidos, no óptimos, basados en compilaciones, que escalen a cuadrículas mucho más grandes. Un ejemplo de tal enfoque es el de Pianpak, Son, Toups, y Yeoh (Pianpak y cols., 2019), que puede escalar a grillas de un tamaño de $100 \times 100$ con más de $1000$ agentes. En su núcleo, Pianpak y cols. (Pianpak y cols., 2019) utiliza una compilación cuadrática de MAPF a ASP, que podría hacerse lineal, y por lo tanto probablemente más rápida, utilizando nuestras ideas.

Otra línea de investigación interesante es la construcción de codificaciones lineales de MAPF a SAT, donde, en base a las mejoras observadas con ASP, esperaríamos mejoras significativas en el rendimiento. Una observación importante, sin embargo, es que las compilaciones ASP son más elegantes que las compilaciones SAT, ya que el proceso de instanciación en ASP se realiza a partir de una representación elevada. En efecto, la codificación de nuestro solver ha sido incluida completamente en esta tesis y no requiere algoritmos de generación de cláusulas que necesitan los solvers basados en SAT.

En la segunda parte de esta tesis propusimos una variante online de nuestra compilación en ASP basada en el uso de ventanas, con el objetivo de poder trabajar en grillas de mayor tamaño. Si bien nuestra evaluación empírica no mostró mejores resultados en términos de rendimiento con respecto al estado del arte, nuestra compilación muestra excelentes resultados en cuando a la suboptimalidad de las soluciones independiente del tamaño de las ventanas. Los resultados confirman y avalan las conclusiones de la primera parte y sugieren seguir con la idea de la construcción de codificaciones de MAPF a SAT para así optimizar los tiempos de instanciación.

# BIBLIOGRAFÍA


Andres, B., Kaufmann, B., Matheis, O., y Schaub, T. (2012). Unsatisfiability-based optimization in clasp. En A. Dovier y V. S. Costa (Eds.), *Technical communications of the 28th international conference on logic programming (iclp)* (Vol. 17, pp. 211–221). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Descargado de `http://drops.dagstuhl.de/opus/volltexte/2012/3623` doi: 10.4230/LIPIcs.ICLP.2012.211

Barták, R., y Svancara, J. (2019). On SAT-based approaches for multi-agent path finding with the sum-of-costs objective. En P. Surynek y W. Yeoh (Eds.), *SoCS* (pp. 10–17). AAAI Press. Descargado de `https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18323`

Barták, R., Zhou, N., Stern, R., Boyarski, E., y Surynek, P. (2017). Modeling and solving the multi-agent pathfinding problem in picat. En *ICTAI* (pp. 959–966). Boston, MA, USA: IEEE Computer Society. Descargado de `https://doi.org/10.1109/ICTAI.2017.00147` doi: 10.1109/ICTAI.2017.00147

Erdem, E., Kisa, D. G., Öztok, U., y Schüller, P. (2013). A general formal framework for pathfinding problems with multiple agents. En *AAAI*. AAAI Press.

Felner, A., Li, J., Boyarski, E., Ma, H., Cohen, L., Kumar, T. K. S., y Koenig, S. (2018, June). Adding heuristics to conflict-based search for multi-agent path finding. En *ICAPS* (pp. 83–87). Delft, The Netherlands. Descargado de `https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17735`

Ferraris, P., y Lifschitz, V. (2005). Mathematical foundations of answer set programming. En S. N. Artëmov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, y J. Woods (Eds.), *We will show them! essays in honour of dov gabbay, volume one* (pp. 615–664). College Publications.

Gebser, M., Kaminski, R., Kaufmann, B., y Schaub, T. (2011). Multi-Criteria Optimization in Answer Set Programming. En J. P. Gallagher y M. Gelfond (Eds.),

*Technical communications of the 27th international conference on logic programming (iclp)* (Vol. 11, pp. 1–10). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Descargado de http://drops.dagstuhl.de/opus/volltexte/2011/3161 doi: 10.4230/LIPIcs.ICLP.2011.1

Gebser, M., Kaminski, R., Kaufmann, B., y Schaub, T. (2014). Clingo = ASP + control: Preliminary report. *CoRR*, *abs/1405.3694*.

Gebser, M., Obermeier, P., Otto, T., Schaub, T., Sabuncu, O., Nguyen, V., y Son, T. C. (2018). Experimenting with robotic intra-logistics domains. *Theory and Practice of Logic Programming*, *18*(3-4), 502–519.

Goldenberg, M., Felner, A., Stern, R., Sharon, G., Sturtevant, N. R., Holte, R. C., y Schaeffer, J. (2014). Enhanced partial expansion A*. *Journal of Artificial Intelligence Research*, *50*, 141–187.

Hart, P. E., Nilsson, N., y Raphael, B. (1968). A formal basis for the heuristic determination of minimal cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2).

Kautz, H. A., y Selman, B. (1992). Planning as satisfiability. En B. Neumann (Ed.), *ECAI* (pp. 359–363). John Wiley and Sons.

Li, J., Felner, A., Boyarski, E., Ma, H., y Koenig, S. (2019). Improved heuristics for multi-agent path finding with conflict-based search. En S. Kraus (Ed.), *IJCAI* (pp. 442–449). ijcai.org. Descargado de https://doi.org/10.24963/ijcai.2019/63 doi: 10.24963/ijcai.2019/63

Lifschitz, V. (2008). What is answer set programming? En *AAAI* (pp. 1594–1597). Descargado de http://www.aaai.org/Library/AAAI/2008/aaai08-270.php

Ma, H., y Koenig, S. (2017). AI buzzwords explained: multi-agent path finding (MAPF). *AI Matters*, *3*(3), 15–19. Descargado de http://doi.acm.org/10.1145/3137574.3137579 doi: 10.1145/3137574.3137579

Pianpak, P., Son, T. C., Toups, Z. O., y Yeoh, W. (2019). A distributed solver for multi-agent path finding problems. En *DAI* (pp. 2:1–2:7). Beijing, China: ACM.

Descargado de https://doi.org/10.1145/3356464.3357702 doi: 10.1145/3356464.3357702

Sharon, G., Stern, R., Felner, A., y Sturtevant, N. R. (2012). Conflict-based search for optimal multi-agent path finding. En *AAAI*. Toronto, Canada.

Silver, D. (2005). Cooperative pathfinding. En *AIIDE* (pp. 117–122).

Standley, T. S. (2010). Finding optimal solutions to cooperative pathfinding problems. En *AAAI* (p. 173-178). Atlanta, Georgia. Descargado de http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1926

Stern, R., Sturtevant, N. R., Felner, A., Koenig, S., Ma, H., Walker, T. T., ... Boyarski, E. (2019). Multi-agent pathfinding: Definitions, variants, and benchmarks. En P. Surynek y W. Yeoh (Eds.), *SoCS* (pp. 151–159). AAAI Press. Descargado de https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18341

Sturtevant, N. (2012). Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, *4*(2), 144 – 148. Descargado de http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

Surynek, P. (2014). Compact representations of cooperative path-finding as SAT based on matchings in bipartite graphs. En *ICTAI* (pp. 875–882). IEEE Computer Society.

Surynek, P. (2019). Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. En S. Kraus (Ed.), *IJCAI* (pp. 1177–1183). ijcai.org. Descargado de https://doi.org/10.24963/ijcai.2019/164 doi: 10.24963/ijcai.2019/164

Surynek, P., Felner, A., Stern, R., y Boyarski, E. (2016). Efficient SAT approach to multi-agent path finding under the sum of costs objective. En *ECAI* (pp. 810–818). IOS Press.

Wang, K. C., y Botea, A. (2008). Fast and memory-efficient multi-agent pathfinding. En *ICAPS* (pp. 380–387). Descargado de http://www.aaai.org/Library/ICAPS/2008/icaps08-047.php

Yu, J., y LaValle, S. M. (2013). Structure and intractability of optimal multi-robot path planning on graphs. En *AAAI*. Bellevue, Washington.