PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

EXPLOITING DIRECTION IN GRID GRAPHS TO BUILD A FAST AND LIGHTER SUBGOAL GRAPH

BRUNO MARÍN BARRERA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JORGE BAIER ARANDA

Santiago de Chile, March 2022

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

EXPLOITING DIRECTION IN GRID GRAPHS TO BUILD A FAST AND LIGHTER SUBGOAL GRAPH

BRUNO MARÍN BARRERA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, March 2022

Gratefully to my parents and siblings

ACKNOWLEDGEMENTS

I would first like to thank my supervisor Jorge Baier, who constantly guided and supported me through this process. Thanks for always having time for talking about my research and our lives.

I would also like to thank Carlos Hernández, who always was aware of my work and provided me important feedback.

I am also grateful to Tansel Uras and Daniel Harabor, for helping me to understand Jump Point Graphs.

I wish to express my sincere thanks to Paulina Peñaloza, for helping me with the writing of this thesis.

I must express my very profound gratitude to my parents for all the support they gave me during my studies and the pandemic. Nothing of this work would be possible without their help, specially my mother's. I would also like to thank my brother, who was always there to listen to my ideas and problems about my research.

Finally, I would like to thank all my friends that helped me to better cope with this process.

Bruno Marín

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In the *path planning* problem on grid graphs, one of the main state-of-art preprocessing techniques are *subgoal graphs*. These graphs consist of a subset of important nodes called subgoals that are connected by a reachability relation. When solving a path planning problem, one must connect the *origin* node and the *destination* node to the subgoal graph, search for a path in the subgoal graph, and refine the path into a path in the original graph. In this thesis, we present Directed Subgoal Graphs (DSG), a new subgoal graph which is built on a grid graph augmented with the incoming direction in order to prune non-shortest paths. The reachability relation ensures that all shortest path and the beginning of any diagonal-first path between two subgoals are unblocked. In DSG the connection between subgoals is performed in a novel cardinal-first order. The connection process shows to be fastest among these in all subgoal graphs while using the minimum amount of memory for this stage.

When DSG is enhanced with Contraction Hierarchies (CH), it improves the state-of-art performance in several instances of the MovingAI benchmark set. In game maps it provides a speedup of up to 3.0% over the state-of-art subgoal graph while using 47% less memory. We observed most of the benefits are obtained in mainly large maps in which the traversable space allows long diagonal movements and there is a large number of isles of obstacles. When this happens, the speedups can reach up to 27%. Additionally, we provide several improvements to the subgoal graph framework, which includes the extension of the avoidance of unimportant subgoals and a new technique to reduce redundant shortcuts generated by CH.

Keywords: **Path planning, Preprocessing based path planning, Grid Graphs, Subgoal Graphs, Jump Point Graphs, Contraction Hierarchies.**

RESUMEN

En el problema de *path planning* sobre grafos tipo grilla, una de las principales técnicas de preprocesamiento del estado del arte son los *subgoal graphs*. Estos grafos consisten en un subconjunto de nodos importantes denominados *subgoals* que son conectados mediante una relación de alcanzabilidad. Al momento de resolver un problema se conecta el nodo origen y el nodo destino al subgoal graph, se realiza una búsqueda en el grafo y se refina el camino obtenido en un camino en el grafo original. En esta tesis, presentamos Directed Subgoal Graphs (DSG), un nuevo subgoal graph que se construye sobre una grilla aumentada con la dirección de incidencia para eliminar caminos que no sean óptimos. La relación de alcanzabilidad asegura que todos los caminos óptimos y el comienzo de algún camino diagonal-first entre dos subgoals sean válidos.
En DSG la conexión entre subgoals se realiza en un orden cardinal-first. El proceso de conexión muestra ser el más rápido respecto al de todos los otros subgoal graphs, al mismo tiempo que necesita la mínima cantidad de memoria para esta etapa. Cuando DSG es potenciado con Contraction Hierarchies (CH), mejora el rendimiento del estado del arte en varias instancias del grupo de benchmarks *MovingAI*. En mapas tipo juegos, nuestro algoritmo es hasta 3.0% más rápido que el mejor de los otros subgoal graphs utilizando un 47% menos memoria. Gran parte de los beneficios se obtienen mayoritariamente en mapas grandes cuyo espacio transitable permite movimientos diagonales amplios donde también hay una gran cantidad de islas de obstáculos. Cuando esto sucede, DSG es hasta un 27% más rápido. Ademas, entregamos mejoras para el conjunto de subgoal graphs, que incluye la extensión de la *evitabilidad* de subgoals poco importantes y una nueva técnica para reducir atajos redundantes generados por CH.

Palabras Claves: **Path planning, Preprocessing based path planning, Grid Graphs, Subgoal Graphs, Jump Point Graphs, Contraction Hierarchies**.

1. INTRODUCTION

The problem of path planning consists of finding a sequence of actions that allows an agent to move from an initial state to a goal state in a given environment. This environment consists of states and actions. States are positions or locations where the agent can stand; whereas, actions are movements the agent can perform to transition between one state to another. Thus, this sequence of actions is a solution to the path planning problem. Environments can belong to different domains where the states and actions represent different real-life elements. There are domains such as road networks and grid maps. In road networks each state is a road junction and each actions consists of movement through a road segment. In this domain, the path planning problem may represent GPS navigation. Grid maps are a discretized two-dimensional space where each possible position or cell can be blocked or unblocked. In this sense, blocked cells stand for obstacles that can not be traversed while unblocked cells represent traversable spaces. A problem in this domain represents navigation while avoiding obstacles.

To solve a path planning problem, the environment is often represented as a graph. In this representation, the environment's states and actions are respectively the nodes and edges in the graph. The solution for this problem is a sequence of edges that connect the initial and goal node i.e. a path in the graph. A problem is often referred to as a query and an algorithm that answers this query is called a search algorithm.

There are environments that are static and known in advance such as road networks or video games. This allows a preprocessing phase to take place that consists in computing valuable information that can be saved and used later in order to speed up the query, these techniques are called preprocessing algorithms. It is possible to classify preprocessing algorithms in domain dependent and domain independent, the latter being used directly on a graph.

The highway dimension of a graph can predict the efficiency of a preprocessing algorithm. When over a graph with small highway dimensions, as in the case of road networks, the speed ups obtained are of several orders of magnitude, with algorithms such as Contraction

Hierarchies (Uras & Koenig, 2018). However, in graphs with larger highway dimensions, as in the case of grid graphs, the speed up obtained is reduced compared to road networks (Abraham, Delling, Fiat, Goldberg, & Werneck, 2016).

In grid graphs, there are many symmetric shortest paths, i.e. paths with the same start and goal that consist of the same series of movements combined in different order (Harabor & Grastien, 2011). This negatively affects the performance of a search algorithm. Therefore, domain preprocessing algorithms often use a mechanism to break the symmetry of shortest paths. Two of these algorithms are Subgoal Graphs (Uras, Koenig, & Hernández, 2013) and Jump Point Graphs (Harabor, Uras, Stuckey, & Koenig, 2019).

These algorithms share a framework that entails (1) in the preprocessing stage building a high-level graph in which a subset of important nodes are connected by a reachability relation, (2) in the query stage (a) connect the start and goal nodes to the graph, then (b) search for a path between those nodes, and finally (c) refine the high-level path into a path in the original graph. This procedure is called *Connect-Search-Refine*.

In SG the important nodes, called subgoals, are located at convex corner cells. Here, the reachability relation ensures all the shortest paths between two reachable nodes are unblocked. On the other hand, JP defines a subgoal graph on the direction extended grid graph, an augmented graph that also considers incoming directions while allowing only shortest and diagonal first paths. As a consequence, there are multiple jump points instead of each subgoal (according to the possible incoming directions) that connect to only diagonal-first reachable nodes.

In certain scenarios, like video games or streets, JP can be up to five 5 times faster than SG. Here, the main differences are that (1) a single JP edge may represent a path of arbitrary length in SG and therefore diminishing solution depths and (2) the edges starting from a given position in JP are splitted into different nodes with different incoming directions and therefore diminishing the branching factor.

This algorithm reveals a new paradigm of subgoal graphs that are also directed. Then, the question arises if we can build other directed subgoal graphs with different properties. Therefore, our hypothesis is as follows: *It is possible to redefine the direction extended* *grid graph to build a new directed subgoal graph, according to a more restrictive reachability relation and in combination with other preprocessing techniques can be faster that other state-of-art subgoal graphs in several benchmarks while also using less memory for the connection phase.*

The contributions of this thesis are:

- **We propose a new subgoal graph called Directed Subgoal Graphs (DSG).** For this purpose, we modify the definition of the *direction-extended grid graph* to only allow shortest paths. Then, we define a subgoal graph over this grid graph using *straight* and *diagonal* subgoals and a reachability relation that ensures all shortest paths and the beginning of a diagonal-first path between reachable subgoals are unblocked.
- In DSG, the connection algorithm uses a novel cardinal-first order to detect reachable subgoals. There, the start and target connection use the same connection schema. This allows **halving the memory used in the connection process in comparison with JP**. The time and space complexity of the preprocessing stage are linear on the grid size while for the forward and backward connect from the query phase are linear in the largest dimension of the grid. Our results show that **DSG has the fastest connection procedure among all subgoal graphs**, achieving an average speedup of 6.5%. However, the complete Connect-Search-Refine procedure is slower than other subgoal graphs.
- **In combination with Contraction Hierarchies, CH-DSG improves the state-of-art of several instances of MovingAI benchmarks.** We obtain a speedup of up to 3.0% over the fastest subgoal graphs in three *games* subcategories and two *rooms* subcategories. Also, CH-DSG uses 47% and 45% less memory in *games* and *all* benchmarks respectively with respect to the state-of-art subgoal graphs. We provide an in-depth analysis of where CH-DSG performs best, and our results show that it is on mainly large maps in which the traversable space allows long diagonal movements and there are a large amount of connected components. In these scenarios, the speedups reach up to 27%.

- **We provide several optimization to the subgoal graph framework.** We extend the avoidance of unimportant subgoals, provide mechanisms to reduce redundant edges in CH and extend the idea of using shortcuts that can be easily unpacked.

This thesis is structured as follows. In Chapter 2 we provide the theoretical framework to this thesis. In Section 2.1 we explain the formal definitions to path planning, graphs and grid graphs. In Section 2.2 we introduce to several search algorithms, explaining their commonalities and differences and in Section 2.3 we provide the basic notions to understand Contraction Hierarchies. In Section 2.4 we introduce to the subgoal framework and in Section 2.5 and Section 2.6 we explain different applications of the subgoal graph framework that produces Subgoal Graphs and Jump Point Graphs respectively. In Chapter 3, Section 3.1 we present Directed Subgoal Graphs (DSG). To do so, we introduce with a motivation with the key ideas behind DSG, then we provide the formal definitions and proofs for it and next we present the algorithms used and their time and space complexities. In Section 3.2, we give and overview of how CH works on the different subgoal graphs and in Section 3.3 we provide several optimizations to the subgoal graph framework. In Chapter 4 we present the experimental evaluation of DSG and all others subgoal graphs. In Section 4.1 We describe the experimental setup. Then, in Section 4.2 we present the results of the preprocessing phase and query phase of all subgoal graphs and their combination with CH applied to the benchmarks. In Chapter 5 we summarize the conclusions and contributions of this thesis.

2. BACKGROUND

In this chapter we first introduce to the formal definitions about graphs, grids, path planning and preprocessing techniques. Then we explain in detail the most used search algorithms, such as BFS, Dijkstra, A* and Bidirectional Dijkstra. Then, we continue explaining Contraction Hierarchies, an *state-of-art* preprocessing technique for graphs. Following, we discuss about the *subgoal graph* framework introduced in (Uras, 2019), followed by the formal definitions and algorithms used in Subgoal Graphs (SG) (Uras et al., 2013) and Jump Point Graphs (JP) (Harabor et al., 2019).

2.1. Problem definition

The problem of path planning consists of finding a sequence of actions that allows an agent to move from an initial state to a goal state in a given environment. This problem is equivalent to find a path in a graph, where states are nodes and actions are edges in the graph. This task can be performed by *search algorithms*, procedures than systematically explore the graph in order to find the sought path.

2.1.1. Graphs

A *weighted* graph is a tuple $G = (V, E, c)$ where V is a set of nodes and E is a set of edges. E is a relation between nodes, i.e. $E \subseteq V \times V$. c is a cost function that maps every edge to a positive real number: $c : E \rightarrow \mathbb{R}_0^+$
G is *undirected* if for every edge $e_1 = (u, v) \in E$ there exist an edge $e_2 = (v, u) \in E$ such that $c(e_1) = c(e_2)$. Otherwise, we say G is a *directed* graph. Also, if $e = (u, v) \in E$ we say e is an *outgoing* edge of v and an *incoming* edge of u. In the same way, we say v is an *incoming neighbour* of u and u is an *outgoing neighbour* of v. For undirected graphs, incoming and outgoing neighbours are the same, therefore we use neighbour.
A path π between v_1 and v_n in G is a sequence of nodes $\langle v_1, v_2, ..., v_{n-1}, v_n \rangle$ such that $(v_i, v_{i+1}) \in E$ for every $i \in \{1, \ldots, n-1\}$. We say a path π pass through a node v if $v \in \pi$.

For two paths $\pi_1 = \langle u_0, ..., u_m \rangle$ and $\pi_2 = \langle v_0, ..., v_n \rangle$ where $u_m = v_0$ the *concatenation* between π_1 and π_2 is denoted by $\pi_1 \cdot \pi_2$ and equals to $\langle u_0, ...u_m = v_0, ..., v_n \rangle$.

The length of a path $\pi = \langle v_1, \ldots, v_n \rangle$, denoted by $l(\pi)$, is the number of edges in it; that is, $l(\pi) = |\pi| - 1 = n - 1$ and the cost of a path $c(\pi)$ equals to the sum of the cost of its edges: $c(\pi) = \sum_{i=1}^{n-1} c(v_i, v_{i+1})$. A path between s and t is called an *s-t* path. A shortest *s-t* path π is such that no other *s-t* path π' exists such that $c(\pi) > c(\pi')$. The *s-t* distance on G equals to the cost of an *s-t* shortest path and is denoted as $d_{s,t}$. If the *s-t* path π is a shortest path it is called an *optimal* path, otherwise it is called an *suboptimal* path. An heuristic, in the context of a graph, is a function $h : V \times V \to \mathbb{R}_0^+$ that estimates the distance between two nodes in the graph. An heuristic is *admissible* if it doesn't overestimate the distance to the destination: $h(u) \leq d_{u,v}$ for all u.

2.1.2. Path planning

The path planning problem between initial and goal states represented by a nodes s and t on a graph G is defined as an s-t *query*. The sought path is also called as an s-t path or as the *solution*, and the solution may be *optimal* or *suboptimal*. An algorithm that always returns an optimal solution, is also *optimal*. If the algorithm outputs suboptimal solutions π and there exists some $w : c(\pi) \leq w \cdot d_{st}$ for all s, t we say the algorithm is *w-suboptimal*. If the solution is optimal, it is an s-t shortest path. Also, we refer the node s as the *start* and t as the *target* or the *goal*.

2.1.3. Preprocessing Based Path Planning

When a graph is known in advance and there is time available before solving the queries, a preprocessing phase can take place. This consists of performing different calculations over the graph with the purpose of producing information that can be used later to speed up the *query*. The set of instructions needed to perform both the preprocessing phase and the query phase is called a *preprocessing algorithm* .

To compare two preprocessing algorithms, the following criteria are taken into consideration: (1) Preprocessing time: Time used in the preprocessing phase to compute the desired information (2) Memory used: The amount of memory that must be saved to store the desired information and (3) Query time: The time needed to solve an s-t *query*.

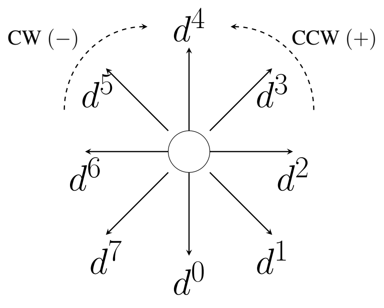
2.1.4. Grid Graphs

Grid graphs are used in navigation tasks over a discretized two-dimensional space, often called as a *map*. A grid graph $G = (V, E)$ is represented as a binary $W \times H$ matrix A, where each entry $A_{x,y}$ can be in one of two possible states: *blocked* or *unblocked*. The set of nodes V consists of *cells* $n = (x, y)$ which are vectors in $\mathbb{Z}^2$ such that $A_{x,y}$ is unblocked. Cells represent position in the space in which the agent can be, using one cell at a time. In a grid graphs, there exists a set of valid *movements* D. A movement is a vector in $\mathbb{Z}^2$ and each edge $(n_1, n_2) \in E$ is such that $n_2 = n_1 + d$ for some direction $d \in D$. Therefore, for a path $\pi = \langle n_1, ..., n_k\rangle$ there exist a sequence of *movements* $\langle d_1, ..., d_{k-1}\rangle$ that holds $n_i + d_i = n_{i+1}, i = 1..k - 1$.

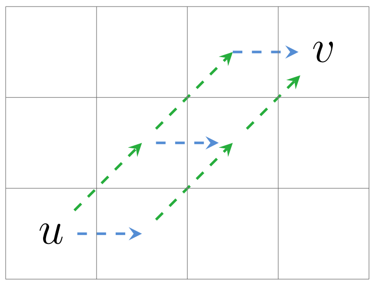
There exist different types of grid graphs depending on the number of valid movements the agent can perform, also called *neighborhood* size. A grid graph with neighborhood size of 4 correspond to a grid graph where the movements are related to one of the four cardinal directions. In the same way, neighborhood size of 8 represents a grid graph where the valid movements are one of the four cardinal or four diagonal directions. It is possible to generalize this idea to build grid graphs with neighborhoods of size 2^k (Rivera, Hernández, Hormazábal, & Baier, 2020), however, we focus our study in grids with neighborhood size of 8 which are *8-connected grid graphs*.

2.1.5. 8-connected Grid Graph

An 8-connected Grid Graph is a grid graph in which there are eight valid movements, each one corresponding to one the following directions.



(a) The eight possible directions.



(b) Freespace u-v shortest paths.

Figure 2.1. (a) Directions with superscripts. Increasing a superscript results in a CCW rotation and decreasing a superscript results in a CW rotation. (b) Any path consisting of two green movements and one blue movement is a freespace shortest path whose cost is the octile distance between u and v.

$$D = \langle(1,0),(1,1),(0,1),(-1,1),(-1,0),(-1,-1),(0,-1),(1,-1)\rangle$$

A *diagonal* direction d satisfies $|d| = \sqrt{2}$ and a *cardinal* direction c is such that $|c| = 1$. *Cardinal* and *diagonal* movements are defined according to their directions.
In some cases, we need to refer correlated directions. For this purpose, we use d^i to refer the i-th direction in the D sequence, regardless if it is a *cardinal* or a *diagonal* one. Also, for $i < 0$ or $i > 8$ we use $d^i = d^{i \bmod 8}$ in order to allow circular references. This way when i is even the direction is cardinal and when i is odd the direction is diagonal. If the superscript is omitted, a cardinal direction is represented by c and a diagonal one is denoted by d. In Figure 2.1 (a) we show the eight possible directions with their respective superscript. Here, increasing a superscript results in a counter-clockwise rotation (CCW-) and decreasing a superscript results in a clockwise rotation (CW+).

The 8-connected graph, whose space is represented by the $W \times H$ binary matrix A, is a graph where $V = \{(x,y) \mid A_{x,y} \text{ is unblocked}\}$ and $E = \{(n_1,n_2) \mid n_1 + d = n_2, \text{ for some } d \in D\}$.

(Harabor et al., 2019) The **freespace graph** for a grid graph G with an associated matrix A is a graph denoted $\mathcal{F}_G$ and is constructed using the matrix $A_{\mathcal{F}}$, which has the same dimensions of A but is such that all positions are unblocked cells. Shortest paths in this graph are called **freespace-shortest-paths**. If a freespace-shortest path π is unblocked on G, then it is also a shortest path on G. The distance between any pair of cells in the freespace is called the *octile distance*. The octile distance between two cells (x_1, y_1) and (x_2, y_2) equals to:

$$\textit{octile\_distance}((x_1, y_1), (x_2, y_2)) = (\sqrt{2} - 1)\min(\Delta x, \Delta y) + \max(\Delta x, \Delta y),$$

where $\Delta x = |x_2 - x_1|$ and $\Delta y = |y_2 - y_1|$. A path whose cost is the octile distance consists of the repetition of two movements, a cardinal and a diagonal one, as shown in Figure 2.1 (b).

In 8-neighbor grid graphs, the most common heuristic function is octile distance. If an heuristic function is not specified, we are referring to the octile distance.

2.2. Search algorithms

A search algorithm over a graph G consists of a sequence of instructions which systematically explores the graph in order to find a path between given pair of nodes. There are *one-to-one*, *one-to-many* and *many-to-many* search algorithms. *One-to-one* refers to algorithms that compute the path between a specific pair of nodes, called *start* and *goal*. *one-to-many*, on the other hand, compute the path from a given *start* node to a subset of G (and potentially G). *Many-to-many* algorithms computes the distances between two subset of G (potentially $G \times G$). A *correct* algorithm is an algorithm that if it finds a solution π, it is a valid path on G. A *complete* algorithm is one that always finds a solution if it exists. Finally, a search algorithm is *optimal* if it always find a path π with $c(\pi) = d_{s,t}$.
The common structure of a search algorithm is shown in Algorithm 2.1. The *Open queue* contains all the *explored* nodes and is used to determine the next node to be *expanded*. For this purpose, it could use a *stack* or a *priority queue*. The *Closed* set contains all already

expanded nodes. The *parents* array keeps the last node $u \neq v$ that passes through an s-v path $\langle s, ..., u, v \rangle$.

Algorithm 2.1 Generic search algorithm.

```
 1: function SEARCH_ALGORITHMM(node $s$, node $t$, graph $G$)
 2:     Open ← $\{s\}$;
 3:     Closed ← $\{\}$;
 4:     parents ← an array with the size of $|V|$;
 5:     while Open is not empty do
 6:         $u \leftarrow$ EXTRACT_NEXT(Open);                    ▷ Expansion of node $u$
 7:         $Closed \leftarrow Closed \cup \{u\}$
 8:         for all $v$ in $u$ outgoing neighbours do;              ▷ Edges relaxation
 9:             if $v \notin$ Open ∪ Closed then
10:                 Open ← Open ∪ $\{v\}$;
11:                 parents[$v$] ← $u$;
12:             else if $v \in$ Open and UPDATE_PARENT($v$, $u$) then
13:                 parents[$v$] ← $u$;
14:     $\pi \leftarrow \langle\rangle$;
15:     $u \leftarrow t$;
16:     while parents[$u$] $\neq s$ do                             ▷ Path reconstruction
17:         $\pi \leftarrow \langle$(parents[u], u)$\rangle \cdot \pi$;
18:         $u \leftarrow$ parents[u];
        return $\pi$;
```

The expansion of a node u (lines 6-13) consists of the *relaxation* of each one of its outgoing edges (lines 8-13) whereas the *edge relaxation* consists of the following steps: In lines 9-11, check if the outgoing neighbour v if it is in the Open. If v is not in the Open, then it is an unexplored node and the path $\langle s, .., u, v \rangle$ is a valid $s - v$ path, therefore, update its parent. In lines 12 and 13, if v is in the Open, then the search algorithm may perform the UPDATE_PARENT procedure. The UPDATE_PARENT, which depends of the specific implementation, consists of checking if the path $\langle s, .., u, v \rangle$ is a path of lower *estimated cost* than the actual $s - u$ path (if exists). If this occurs, the $s - v$ path is updated by line 13 such that it now passes through u. The estimated cost is also defined in each implementation. The algorithm ends with the *path reconstruction* (lines 14-18), which consists of building the path from the end to the beginning using the parents array.

Now we prove correctness of this algorithm. If, when given s and t, we show that π is an s-t path. The returned sequence π consists only of (parent[u], u) tuples. Also, (parent[u],

u) is an edge since it is defined in lines 8, 11 and 13. Therefore π is a path on G. Now we show that π is an s-t path. For line 1, s is the first node to be expanded and also is the first node in every path. For line 15, the last edge of π ends in t. Therefore π is an s-t path and 2.1 is correct.

Next we prove that this algorithm is complete. Since each expanded vertex is added to the Closed (line 7) and each node is added to the open only if it isn't in the *Open* or *Closed* already, this algorithm expands every node at most once. Also, for line 8, each node relaxes its edges at most once, therefore, every edge is checked at most once. Thus, if G is finite, the algorithm will halts.

In the expand node phase, the algorithm performs $O(|V|)$ *Open* queue deletions and *closed* insertions. In the *Edges relaxation phase*, it performs $O(|E|)$ queue insertions and the same order of *Closed* checks, *Open* checks and UPDATE_PARENT calls.

A search algorithm can be seen as the exploration of *search tree*. The number of expanded nodes during a search algorithm is the *search space*. The branching is the average number of nodes that are added to the Open after an expansion. Finally, the *solution depth* is the length (number of edges) of the returned path π. Specific details of the most common search algorithms are presented as follows.

2.2.1. BFS

Breadth first search (BFS) uses a first-in first-out queue. Given it uses a *breath* exploration, BFS is guaranteed to find path with the minimum amount of edges. In this algorithm the queue insertion and deletion are $O(1)$, resulting in a $O(|V| + |E|)$ time complexity.

2.2.2. Dijkstra

Dijkstra uses a *priority* queue in which the priority of a node u is the distance $d_{s,u}$. The priority, called g, is initialized as $g(v) = \infty$ for all $v \in G \setminus \{s\}$ and $g(s) = 0$. The

g-value of a node v explored from u node equals to $g(v) = g(u) + c((u, v))$. In Dijkstra, the UPDATE_PARENT returns *true* when $g(v) > g(u) + c((u, v))$ and *false* otherwise, i.e. u is being discovered from path of lower cost that pass by v. Dijkstra is guaranteed to find the shortest s-t path and is used in *one-to-many* queries.

2.2.3. Bidirectional Dijkstra

Bidirectional Dijkstra presents a new approach to solve an s-t problem that consists of performing simultaneously a search from the *start* and another search from the *goal*. The objective is to explore two search spaces with half radius, resulting in a smaller search space with respect to *Dijkstra*. These searches explore the *forward* and *backward* graph respectively. The *forward* graph $G_f = (V, E_f, c)$ equals to G, but for clarity we add the f sub index. The *backward* graph $G_b = (V, E_b, c)$ is a graph where for each (u, v) in E_f there is $(v, u) \in E_b$ with $c((u, v)) = c((v, u))$. We use f and b sub indices to distinguish between G, E, *Open* , *Closed*, g from *forward* and *backward* graphs and searches.
To simulate *simultaneous* searches, the algorithm decides which open should be expanded in every iteration using a criterion. The most common criteria are: (1) Expand the queue with the lowest g value (2) Expand the queue with maximum size (3) Alternate between Open$_f$ and Open$_b$.
The algorithms finishes after (1) the two frontiers meet at node u and (2) only paths with cost higher than $g_f(u) + g_b(u)$ can be found. Bidirectional Dijkstra is used to solve *one-to-one* queries and it is based on that the area of a circle with radius r is larger than the area of two circles of radius $\frac{r}{2}$.

2.2.4. A*

A* uses a priority queue with a priority term defined as $f(u) = g(u) + h(u)$. g is the same function used in Dijkstra while h is an *heuristic* function. If A* is run with an *admissible* heuristic, it is guaranteed to find an *optimal* solution. In this algorithm g and UPDATE_PARENT are used in the same way than in Dijkstra. If the priority queue is implemented used a *Min Heap*, then the worst case time complexity of A* is $O((|V| + |E|)log_2|V|$.

2.3. Contraction Hierarchies

Contraction Hierarchies (CH) (Geisberger, Sanders, Schultes, & Delling, 2008) is a preprocessing algorithm that builds an augmented graph $G' = (V, E')$. In this graph, there are additional edges between nodes, called *shortcuts*. Shortcuts are used to speedup queries in the graph, however, they should be added smartly to avoid graphs with size $V \times V$. For this, CH performs and ordering of all nodes where nodes are sorted from the least *important* to the most *important*. Then, every node is *contracted* according that order. Once a node is contracted, it is assigned with a *hierarchy level*, which must be higher than the level of other already contracted nodes. The preprocessing phase ends when all nodes are *contracted*.

Contraction is the process by which a node $v \in V$ is temporarily removed of the graph. In order to preserve shortest paths in the remaining graph, shortcuts between incoming neighbours u and outgoing neighbours w are added, but only if the path $\langle u, v, w \rangle$ is the unique u-w shortest path. Also, the node and all its edges are removed temporarily from the graph.

In order to determine the shortcuts that need to be added into the graph after in case of a contraction of a node v, a modified Dijkstra is performed from every incoming neighbour u. This search ends after all outgoing neighbours w of v are expanded. Also, it does not add v to the Open (Algorithm 2.1, line 9) in order to verify if the only shortest path is $\langle u, v, w \rangle$. The execution of this version of Dijkstra is the most time-consuming task in the preprocessing phase, because of this, it may be useful to limit the search space of these searches. The *search space limit* (L) is a parameter of CH and is used to improve the preprocessing times.

In the aforementioned procedure, the *importance* of a node plays a crucial role, since it determines the ordering of the nodes, and therefore the final size of the augmented graph. Thus, the ordering of the node must minimize the shortcuts added to the graph at the same time it improves the average query speed. The importance of a node is represented as a linear combination of different *priority terms*, some of them are described below.

- **Edge difference (E):** The *edge difference* in the contraction of a node v is arguably the most important priority term. It represent the number of shortcuts that need to be added to G in order preserve shortest paths in the remaining graph minus the sum of the incoming and outgoing edges from v.
- **Deleted/contracted neighbors (D):** Is the number of neighbours that are already contracted. This term is used to contract nodes uniformly on the grid, this way improving query speed.
- **Search space (S):** It is a measure of the cost of a contraction. It equals to the sum of the search space of all Dijkstras associated to the contraction of v. It can be used to decrease preprocessing time.

The importance of a node, also referred as the *priority*, can be determined in an initial phase of CH, however, subsequent node contractions can change its value. Therefore, determining an optimal ordering of the nodes can be a time-consuming task. To deal with this, there are heuristics that can be used to reduce preprocessing time at the cost of obtaining sub-optimal ordering. One of these heuristics is presented in (Geisberger et al., 2008). This consists of (1) before contracting a node, recalculate its priority to verify that it is the least important node, otherwise, repeat this process with the new least important node. (2) each time a node is contracted, update the priority of its neighbours, since their priority probably changed. (3) Periodically update all priorities.

Once the hierarchy is built, an s-t query can be solved by performing a bidirectional search starting from s and t. In those searches, only edges that go up in hierarchy are used, therefore, reducing substantially the branching factor. Also, shortcuts help reaching

solutions with lower depth. After the bidirectional search is finished, the resulting path may consist of edges on the original graph and shortcuts. In order to have a path in the original graph, every shortcut must be *unpacked* into the corresponding edges on G.
One way to implement the *unpacking procedure* is by *2-pointer unpacking*. For performing this procedure, the two *base* edges e_1 and e_2 that composes a shortcut must be saved. This edges can be shortcuts too. Then, for unpacking a shortcut it is necessary to recursively unwrap both of its edges in its corresponding base edges, until these edges are not shortcuts.

2.4. The Subgoal Graph framework

The *Subgoal Graph Framework* (Uras, 2019) is a general idea used in pathfinding that uses preprocessing to speed up pathfinding. During preprocessing time, we are given the search graph G, and we build another graph G_S, whose nodes are a subset of the nodes of G. During query time, we are given a start and a goal node of G and find a path connecting such nodes by carrying out search over G_S.

Both Subgoal Graphs (Uras et al., 2013) and Jump Point Graphs (Harabor et al., 2019) fit into the subgoal graph framework. In the rest of the section, we give definitions that are common to these two graphs, and which are also be the basis for directed subgoal graphs, the type of graphs we propose in this thesis.

2.4.1. Formal definition

We start off with some of the the basic definitions needed to compute a subgoal graph. All definitions in this section were first proposed in (Harabor et al., 2019).

Definition 2.1. *Given a graph $G = (V, E)$, a reachability relation R is a relation such that $R \subseteq V \times V$, and:*

(i) *For every $v \in V$, $(v, v) \in R$.*

(ii) *For every $(v_1, v_2) \in E$, $(v_1, v_2) \in R$.*

When $(v_1, v_2) \in R$ we say v_2 is R-reachable from v_1 or v_1 R-reaches v_2. The reader may notice at this point that our definition of reachability relation is very general. For example, we may say that v_1 and v_2 are horizontally reachable iff (1) $v_1 = v_2$, (2) $v_2 = v_1 + d$, for some $d \in D$, or (3) there is a path from v_1 to v_2 generated by performing cardinal move $(1, 0)$. Not all reachability relations allow us to define useful subgoal graphs. We give more examples in the following sections.

Definition 2.2. *t is direct-R-reachable (DR-reachable) from s with respect to $S \subseteq V$ on G iff $(s, t) \in R$ and no shortest s-t path on G passes through any $v \in S$.*

For an example of direct-R reachability, consider once again horizontal reachability, defined above, and let $S = V$. Observe that if n_1, n_2, and n_3 are all horizonatally reachable from each other, but a shortest path from n_1 to n_3 passes through n_2, then n_3 is not direct-horizontally reachable from n_1. Direct reachability is later used to define the connections in subgoal graph. We aim these connections to be as fewer as possible. By using direct reachability we avoid adding arcs that are not necessary for search.

Below we may say simply that 't is DR-reachable from s' when set S is clear from the context.

Definition 2.3. *$S \subseteq V$ is an $R-$shortest-path-cover ($R-$SPC) on G iff, for all $s, t \in V$ if $(s, t) \notin R$, then at least one shortest s-t path on G passes through some $v \in S$*

When a set of nodes is a shortest-path-cover we know that no optimal paths between nodes of the cover are 'lost'. Indeed, we define a subgoal graph as one in which such shortest paths are maintained, given a specific reachability relation R, as follows.

Definition 2.4. *$G_S = (S, E_S)$, where $S \subseteq V$ is a set of subgoals, is a subgoal graph on G with respect to R iff S is an R-SPC on G and for all $u, v \in V$ such that $u \neq v$, it holds that $(u, v) \in E_s$ iff v is DR-reachable with respect to S from u.*

2.4.2. Preprocessing algorithm

2.4.2.1. Building the graph

In order to compute a subgoal graph according to Definition 2.4, a set of subgoals S and a reachability relation R are needed. However, it is possible to describe high level algorithm that assumes that those elements are given, as shown in Algorithm 2.2. The input of this process is the grid graph represented by its binary matrix A whose dimensions are $W \times H$.

The function DEFINE_SUBGOALS invokes the IDENTIFY_SUBGOALS function for every unblocked cell n whereas the latter function adds subgoals associated with n to the set of subgoals S. Here, we use the word 'associated' since, as explained in Section 2.5 and Section 2.6, subgoals can be cells $(v \in V)$ or tuples of cells and directions $(v \in V \times D)$.

The function CONNECT_GRAPH receives the set of subgoals S and for each subgoal v calls the FORWARD_CONNECT for v, which return all DR-reachable subgoals v' from v.

Finally, once the subgoal graph G_S is computed by invoking COMPUTE_SUBGOAL_GRAPH, it can be stored to be used later, in the query phase, to speedup an $s - t$ query.

2.4.2.2. Clearances

The FORWARD_CONNECT procedure, which consists of finding every DR-reachable subgoals v' from every subgoal v, can be quite expensive if performed in a naive manner. Given this, it is often optimized using the concept of *clearances*.

Formally, the clearance $C[n, d^i]$ is a function that returns how many movements are needed to move from cell n in direction d^i to reach the nearest *important cell* n' if exists, otherwise returns zero.

Important cells can be (1) cells that contain subgoals that satisfy a given property (*simple important*) or (2) cells that *lead* to another simple important cell along a given

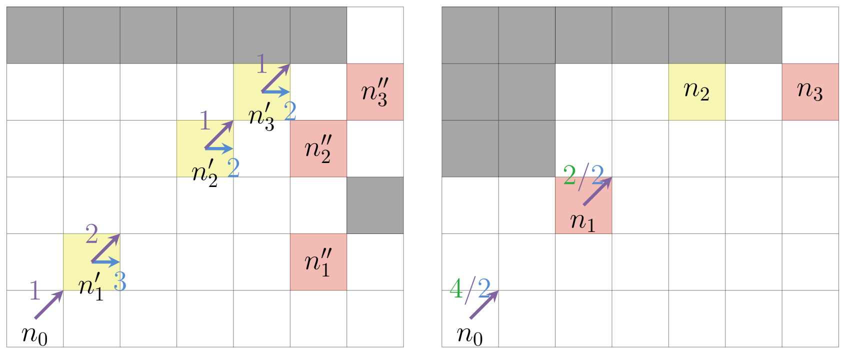
Algorithm 2.2 Constructing a generic subgoal graph from a binary matrix A. Blue and red lines are subgoal graph-specific.

```
1:  function DEFINE_SUBGOALS($A$)
2:      $S \leftarrow \emptyset$ ;
3:      for all *unblocked* cells $n$ from $A$ do
4:          IDENTIFY_SUBGOALS($n, S, A$);
5:      return $S$
6:  function CONNECT_GRAPH($S$)
7:      $E_S \leftarrow \emptyset$;
8:      for all $v \in S$ do
9:          $E_v \leftarrow$ FORWARD_CONNECT($v$);
10:         for all $v' \in E_v$ do
11:             $E_S \leftarrow E_S \cup \{(v, v')\}$;
12:     return $E_S$
13: function BUILD_SUBGOAL_GRAPH(A)
14:     $S \leftarrow$ DEFINE_SUBGOALS($A$);
15:     $E_S \leftarrow$ CONNECT_GRAPH($S$);
16:     $G_S \leftarrow (S, E_S)$
17:     return $G_S$
```

direction (*recursive important*). We say that an important cell n' *leads* to another important cell n'' along d iff $n' + k \cdot d = n''$, for some $k > 0$ with $k \in \mathbb{N}$.

This way, a clearance that uses a simple important cell is a *simple clearance*. A *recursive clearance* $C[n, d^i]$ is a clearance that uses a recursive important cell that leads to n'' where n'' must be an important cell for a different simple clearance. Therefore, the recursive clearance that reaches n' must satisfy $C[n', d] > 0$ for another simple clearance. An hybrid clearance uses both types of important cells. Diagonal and cardinal clearances are defined according to their directions.

Most simple important cells are located at *convex corner cells*. A convex corner cell is an unblocked cell n which satisfy (1) for d^i a diagonal direction, $n + d^i$ is blocked and (2) $n + d^{i+1}$ and $n + d^{i+1}$ are unblocked. An example of the different clearances using convex corner cells as simple important cells is shown in Figure 2.2. Despite this, each subgoal graph must define its own clearances.



(a) Simple and recursive clearances. (b) Recursive and hybrid clearances.

Figure 2.2. (a) The diagonal clearance towards NE is recursive, i.e., it measures distance to the nearest recursive important cell (yellow). The cardinal clearance towards E is simple, i.e., it measures distance to the nearest simple important cell (red). Using these clearances is possible to determine a path from n_0 to n''_1, n''_2 and n''_3 checking only 3 recursive and 3 simple clearances. (b) If the clearance towards NE is recursive, the value from n_0 is 4, reaching n_2. However, if the clearance is hybrid, the value from n_0 is 2, reaching n_1. These clearances are equivalent from n_1, both reaching n_2.

2.4.3. Query algorithm

Once the graph is built, the procedure to solve an s-t query is called *Connect-Search-Refine* **[CSR]** and is explained as follows:

(i) The first step is to **try a direct path**. This consists of checking if an s-t freepsace shortest path is unblocked on G. If a direct path is found, then it is the shortest path and this complete procedure ends. The worst case time complexity is $O(W + H)$, however, since finding any blocked cell interrupts this process, the time consumed in this phase is low.

(ii) **Connect**: Then, only if the start does not already belong to S, it must be connected to other DR-reachable subgoals using the *forward connection*. In the same way, only if the target does not already belong to S, all subgoals that DR-reach the target must connect to it. The latter procedure is referred as *backward* *connection*. The complexity of the connect stage depends on the subgoal graph implementation.

(iii) **Search**: Next, a search in the subgoal graph is performed. Any search algorithm could be used, but best first ones such as A* are the preferred choices. Once the search ends, it returns π_S which is a path on the subgoal graph. The complexity of A* is $O((\mid S \mid + \mid E_S \mid) \cdot \log_2 \mid S \mid)$.

(iv) **Refine**: Here we convert π_S into a path on G. If R is a relation that ensures that at least one freespace shortest path is unblocked on G, then the refinement process consists of:
Let $\pi_s = \langle n_0, ..., n_k \rangle$ be the s-t shortest path on G_s where $n_0 = s$ and $n_k = t$. For each pair of subsequent subgoals n_i and n_{i+1} let π_i be *any* n_i-n_{i+1} freespace shortest path on G. Next, let π'_i be the subpath without its last cell. Finally the concatenation of $\pi'_0 \cdot ... \cdot \pi'_{k-1} \cdot t$ is a valid shortest path on G. This procedure has a linear complexity on π_s cost.

For each subgoal graph instance we must define the forward connection only if it differs from the preprocessing forward connection and the backward connection only if it differs from the query forward connection. Also, we must define the default freespace shortest path used in the refine procedure.

2.5. Subgoal Graphs

Subgoal Graphs (**SG**) (Uras et al., 2013) is one of the type of graphs that is generalized by the subgoal graph framework.

2.5.1. Formal definition

Our definitions are adapted from (Uras et al., 2013) and (Harabor et al., 2019).

Definition 2.5 (Subgoals)**.** *An unblocked cell $n \in V$ is a subgoal iff there exists a diagonal direction d^i such that $n + d^i$ is blocked and $n + d^{i+1}$ and $n + d^{i-1}$ are unblocked.*

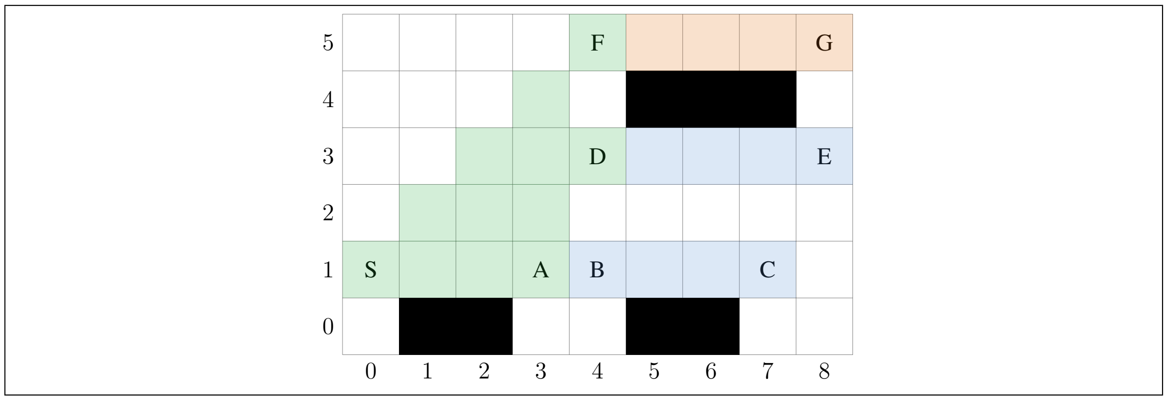


Figure 2.3. Subgoals are marked with letters. A, D and F are DSFR from S. B and C are SFR from S but not direct since A is in the S-B and S-C path. E is also SFR from S but not direct since D is in a S-E path. Finally, G is FR from S since there is an unblocked freespace path, but is not SFR since there are subgoals and obstacles in other freespace paths.

Definition 2.6 (Safe-Freespace reachability)**.** *Two cells are safe-freespace-reachable **[SFR]** if each freespace-shortest path between them are is also a path on G.*

Below we use the abbreviation **DSFR** to refer to direct-safe freespace reachability. Figure 2.3 shows an example of the different reachability relations.

Figure 2.3 shows an example of the different reachability relations.

In (Uras, 2019) the authors proved that the set of subgoals is a SFR-SPC on G. Therefore one can build a subgoal graph $G_S = (S, E_S)$ where S is the set of subgoals and R is safe-freespace-reachability. This subgoal graph is called Subgoal Graph **[SG]** since it is the former subgoal graph.

2.5.2. Framework implementation

Using Definition 2.5 and Definition 2.6 we can instantiate a subgoal graph. The next step is to implement the procedures of the subgoal graph framework.

2.5.2.1. Identify subgoals

In the IDENTIFY_SUBGOALS is implemented in Algorithm 2.3 according to Definition 2.5. It is an straightforward implementation which perform a for loop with constant size and constant-time operations, therefore it is $O(1)$ in time and space.

Algorithm 2.3 SG: Identify subgoals in cell n.

```
1: function IDENTIFY_SUBGOALS($(n, S, A)$)
2:     for all diagonal directions superscripts $i$ do
3:         if $n + d^i$ is blocked on $A$ then
4:             if $n + d^{i+1}$ and $n + d^{i-1}$ are unblocked on $A$ then
5:                 $S \leftarrow S \cup \{n\}$;
```

2.5.2.2. Clearances

The clearances used to detect DSFR subgoals are

- Cardinal clearances $C[n, d]$: These are simple clearances that reach any subgoal along d.
- Diagonal clearances $C[n, d^i]$: These are hybrid clearances that reach (1) any subgoal along d^i and (2) cells n' with $C[n', d^{i\pm1}] > 0$.

2.5.2.3. Forward connection

In order to construct the set of edges, we must define the FORWARD_CONNECT which performs the detection of DSFR subgoals. This is implemented in Algorithm 2.4, which is a modified version of the algorithm presented in (Harabor et al., 2019) that includes an adaptation of the optimization for allowing only direct connections presented in (Uras et al., 2013).

In this algorithm, since subgoals are cells, we use n instead of v to refer to them. The core idea is to perform a systematic exploration of the space using diagonal-first scans from the cell n at the same time it tracks the cardinal distance to subgoals in order to preserve DSFR connections. Diagonal-first scans (DF-scans) are procedures in which a diagonal

and recursive clearance is checked iteratively, while checking the corresponding simple and cardinal clearances in every step.
In detail. the function CARDINAL_SCAN performs the connection to DSFR subgoals checking a cardinal clearance while respecting and returning the new boundaries for direct connections (lines 3, 5 and 6). Since it consists only of constant-time operations it is $O(1)$ in time and space.
The function FORWARD_CONNECT performs the connection to (1) DSFR subgoals that are reached by a cardinal path (lines 10-11) (2) DSFR subgoals that are reached by a diagonal path (lines 19-21) and (3) DSFR subgoals that are reached by a path with diagonal and cardinal movements (lines 12-24). The latter procedure is performed using DF-scans. The time and space complexity of FORWARD_CONNECT is as follows: First, the time complexity of CARDINAL_SCAN is $O(1)$ since it only performs constant-time operations. Then, the for loop in lines 10-11 has constant size and performs only constant-time operations, therefore it is $O(1)$. The for loop in lines 12-24 has constant size but the while loop in lines 17-24 iterates through a diagonal axis of the grid. Since each operation performed inside this loop is constant, the time and space complexity of Algorithm 2.4 is linear on the smallest grid dimension.

After the subgoal graph is built, a s-t query can be answered using the CSR procedure. In Subgoal Graphs, the forward and backward connection are equivalent.

2.5.2.4. Refine

The refinement process of the CSR procedure can be performed by using any freespace-shortest path, since SFR guarantees all those paths are unblocked. To simplify this process we use paths which prioritize diagonal movements before cardinal movements.

Algorithm 2.4 SG forward connection from a cell n.

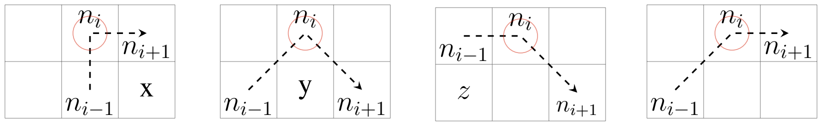
```
function CARDINAL_SCAN($n, i, b$)
    $clr \leftarrow C[n, d^i]$;
    if $clr > 0$ and $clr < b$ then
        $E_n \leftarrow E_n \cup \{n + clr \cdot d^i\}$;
        return $clr$;
    return $b$;
function FORWARD_CONNECT($n$)
    $B \leftarrow$ array of size 8;                                ▷ Cardinal boundaries
    $E_n \leftarrow \emptyset$;
    for all $i$ cardinal direction superscripts do              ▷ Cardinal connection
        $B[i] \leftarrow$CARDINAL_SCAN($n, i, \infty$);
    for all $j$ diagonal direction superscripts do
        $b^- = B[j-1]$;
        $b^+ = B[j+1]$;
        $clr \leftarrow C[n, d^j]$;
        $n' \leftarrow n$;
        while $clr > 0$ do                                        ▷ DF-scans
            $n' \leftarrow n' + clr \cdot d^j$;
            if IS_SUBGOAL($n'$) then                              ▷ Diagonal connection
                $E_n \leftarrow E_n \cup \{n\}'$;
                break;
            $b^- \leftarrow$CARDINAL_SCAN($n', j-1, b^-$);
            $b^+ \leftarrow$CARDINAL_SCAN($n', j+1, b^+$);
            $clr \leftarrow C[n', d^j]$;
    return $E_n$;
```

2.6. Jump Point Graphs

Jump Point Graphs (JP) (Harabor et al., 2019) is a preprocessing algorithm that builds a subgoal graph over the *direction-extended grid graph*. It formalizes the Jump Point Search search space as a graph, defining a set of *jump points* and the *direct diagonal first freespace reachable* reachability relation to define the edges of the graph.

2.6.1. Jump Point Search

Jump Point Search **(JPS)** (Harabor & Grastien, 2011) is an online search algorithm, i.e., it requires no preprocessing. It is based on exploring only-diagonal first paths, while only expanding a subset of the explored nodes called *jump points*. These *jump points* can



(a) card-to-card (b) diag-to-diag (c) card-to-diag (d) diag-to-card

Figure 2.4. Examples of the four types of turning points and its respective conditions to belong to shortest and DF paths. (a) A card-to-card turning point can only belong to a DF and shortest path if x is blocked. (b) A diag-to-diag turning point can not belong to any shortest path, since if y is blocked the diagonal movement is not possible and otherwise it would not be a shortest path. (c) A card-to-diag turning point can belong to a shortest path and in order to be in a DF path z must be blocked. (d) A diag-to-card turning point can belong to DF and shortest paths.

be (1) cells that can be used to circumnavegate an obstacle or (2) cells from which a repeated cardinal movement leads to the former case. Now we present the formal definitions of JPS that are transcendental to the understanding of the search space that is generated. These definitions were obtained from (Harabor & Grastien, 2011).

Definition 2.7. *A turning point is any node n_i along a path $\pi = \langle n_1, n_2, \ldots, n_k \rangle$ where the direction of travel from the previous node n_{i-1} to n_i is different to the direction of travel from n_i to the subsequent node n_{i+1}, that is, $n_i - n_{i-1}$ and $n_{i+1} - n_i$ have different directions. A turning point n_i is* card-to-diag *if the direction of $n_i - n_{i-1}$ is cardinal and the direction of $n_{i+1} - n_i$ is diagonal. A turning point n_i is* diag-to-card *if the direction of $n_i - n_{i-1}$ is diagonal and the direction of $n_{i+1} - n_i$ is cardinal.*

Definition 2.8. *A path π is diagonal-first (DF) if it contains no card-to-diag turning point $\langle n_{k-1}, n_k, n_{k+1} \rangle$ which could be replaced by a diag-to-card turning point $\langle n_{k-1}, n'_k, n_{k+1} \rangle$ to produce a new valid path on G.*

Definition 2.9. *A path π is cardinal-first (CF)) if it contains no diag-to-card turning point $\langle n_{k-1}, n_k, n_{k+1} \rangle$ which could be replaced by a card-to-diag turning point $\langle n_{k-1}, n'_k, n_{k+1} \rangle$ to produce a new valid path on G.*

In Figure 2.4 we show all different turning points and explain its respective conditions to belong to shortest and DF paths. A Diag-to-diag turning point can not belong to a shortest path, whereas a card-to-card can only belong to a shortest and DF path if surrounding an obstacle. A Card-to-diag turning point can belong to a shortest path, but in order to belong to a DF path it must surround an obstacle, whereas a diag-to-card turning point can belong to a shortest and DF path without extra conditions.

During a search, JPS only adds nodes to the Open list that reached by a diagonal-first path. Also when expanding a node, it scans along continuations of the possible diagonal first paths and only take as successors a subset of nodes, called *jump points*, that generates different branches of diagonal-first paths. The successors are called *jump points* and there are 2 types of them, that are mentioned below. This definitions are extracted from (Harabor et al., 2019)

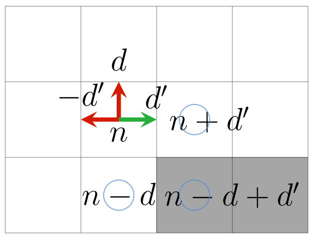
Definition 2.10. *A straight jump point is a tuple (n, d) where d is a cardinal direction and for some $d' \perp d$, $n + d'$ is unblocked, $n - d$ is unblocked and $(n - d) + d$ is blocked.*

Figure 2.5(a) illustrates this definition, while also it shows that a single convex corner cell can generate up to two straight jump points in a cell.

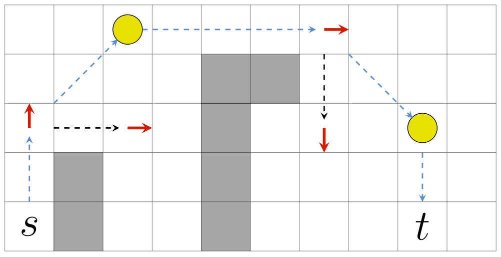
Definition 2.11. *A diagonal jump point is a tuple (n, d) where n is a cell and $d = c_1 + c_2$ is a diagonal direction such that for some direction $c \in \{c_1, c_2\}$ and $k \in \mathbb{N}$ is true that the freespace shortest path between n and $n' = n + k \times c$ is unblocked and n' is a straight jump point or n' is the target.*

In order to reach the target, it is also generated as a successor, according to Definition 2.11. In Figure 2.5(b) an execution of JPS for solving an s-t problem is shown. We observe that only a small subset of nodes is expanded, resulting in faster query times with respect to A*.

In (Harabor & Grastien, 2011) they proved JPS is optimal. Since it also returns only diagonal-first shortest paths, then it is possible to establish the following lemma.



(a) A straight jump point (n, d).



(b) s-t query using JPS.

Figure 2.5. (a) In order to define a jump point in (n, d), two adjacent cells $n + d'$ and $n - d$ must be unblocked and $n - d + d'$ must be blocked. One can notice that $(n, -d')$ is also a jump point. (b) The red arrows and yellow disks represent the expanded straight and diagonal jump points respectively. The dashed blue line represents the actual path and the dashed black line represent other explored jump points.

Lemma 2.1. *(Harabor et al., 2019) For every shortest path π on G, there exists a path π' on G which has the same cost and is diagonal-first.*

A drawback of JPS is that since it does not explicitly build a graph, it cannot be enhanced with other preprocessing algorithms such as Contraction Hierarchies. For this reason, the question arises whether it is possible to represent the search space of JPS as a subgoal graph. The formalization of this idea is presented bellow.

2.6.2. Formal definition

The formalization of JPS search space is presented below. The following definitions and lemmas are obtained from (Harabor et al., 2019).

Definition 2.12. *A path $\pi = \langle n_1, ..., n_k \rangle$ is taut iff every subpath of the form $\langle n_{i-1}, n_i, n_{i+1} \rangle$ of π is a shortest path between n_{i-1} and n_{i+1}.*

Definition 2.13. *Let $G = (V, E)$ be a grid graph. The corresponding direction-extended grid graph $G^* = (V^*, E^*)$ is as follows*

- $V^* = \{(n, d) \mid n \in V, d \in D\}$.

- *For each* $(n_1, n_2) \in E$ *and each pair of grid moves* d_1 *and* d_2 *there exists* $((n_1, d_1), (n_2, d_2))) \in E^*$ *if and only if:*
  (i) $n_1 + d_2 = n_2$
  (ii) *if* $\langle (n_1 - d_1), n_1 \rangle \in E$ *then* $\langle n_1 - d_1, n_1, n_2 \rangle$ *is diagonal-first and taut; otherwise,* $d_1 = d_2$.

Intuitively, being at node (n, d) in the direction-extended graph should be interpreted as "we are at node n *while coming* from another node in G by applying movement d". This is established by Condition 1 in Definition 2.13. Condition 2 of Definition 2.13 establishes that connections cannot be produced between nodes if such connections would allow the existence of diagonal-first and taut path. Notice that if $n_1 - d_1$ does not exist—this could be due to the fact that n_1 is at the border of the grid or adjacent to an obstacle—we only ask that $d_1 = d_2$ to be consistent with the fact that connections in G^* reflect only shortest-path connections.

G^* can be seen as G with two main differences: (1) Each cell is augmented with the possible incoming directions (2) Any edge $((n_1, d_1), (n_2, d_2)))$ also considers the cell $n_1 - d_1$ and the path defined by $\langle n_1 - d_1, n_1, n_2 \rangle$ must be diagonal-first and taut. Therefore, even if the set of nodes is larger, the set of edges is reduced. Also, as shown in the following lemma, the number of paths is also reduced to only allow diagonal-first and taut paths.

Lemma 2.2. *Any path* π^* *on* G^* *is diagonal-first and taut.*

The Lemma 2.2 shows that G^* can be used to solve a path planning problem by treating all (s, d) as start nodes and all (t, d) as goal nodes, for all $d \in D$. Also, it is possible to notice that all straight jump points belong to G^*. Therefore, in order to construct a subgoal graph on G^* it is necessary to define a rechability relation those jump points.

Definition 2.14. *The diagonal-first freespace-shortest path from* n *to* n' *is a freespace-shortest path where all diagonal moves appear before cardinal ones. A node* $(n', d') \in V^*$

*is a diagonal-first freespace reachable **[DFFR]** from $(n, d) \in V^*$ iff a path from (n, d) to (n', d') on G^* corresponds to the diagonal-first freespace-shortest path from n to n'. We use **DDFFR** to denote direct-DFFR reachability.*

Now we prove the following theorem, whose proof was presented in (Harabor et al., 2019).

THEOREM 2.1. *Straight jump points form a DFFR-SPC on G^**

PROOF. Let $s^* = (s, d)$ and $t^* = (t, d)$. For a contradiction, we assume $(s^*, t^*) \notin$ DDFFR and no shortest path passes through a jump point. By Lemma 2.2 π^* is diagonal-first and taut, therefore π^* must have at least one turning point. If a turning point is card-to-card, then it either pass through a jump point or it is not taut. If a turning point is diag-to-diag it is not taut. If a turning point is card-to-diag it is either not locally diagonal-first or it passes through a jump point. Finally, if there exists only one turning point and it is diagonal-to-cardinal then $(s^*, t^*) \in$ DDFFR. Thus, either (s^*, t^*) are DDFFR or π^* pass through a jump point. □

For Theorem 2.1 we can define the subgoal graph G_S with S as the set of straight subgoals and DDFFR as the reachability relation that generates E_S. This subgoal graph is called Jump Point Graph **[JP]**.

2.6.3. Framework implementation

We now define the procedures in the subgoal graph framework.

2.6.3.1. Identify subgoals

The IDENTIFY_SUBGOALS function is implemented in Algorithm 2.5. It consists of adding the corresponding two straight jump points to S according to Definition 2.10, but only if they do not already belong to it. This algorithm is $O(1)$ in space, and if the set operations are $O(1)$, the complexity of this process is $O(1)$ in time too.

Algorithm 2.5 JP: Identify subgoals in cell n.

```
1: function IDENTIFY_SUBGOALS($(n, S, A)$)
2:     for all diagonal directions superscripts $i$ do
3:         if $n + d^i$ is blocked on $A$ then
4:             if $n + d^{i+1}$ and $n + d^{i-1}$ are unblocked on $A$ then
5:                 if $(n, d^{i-3}) \notin S$ then
6:                     $S \leftarrow S \cup \{(n, d^{i-3})\}$;
7:                 if $(n, d^{i+3}) \notin S$ then
8:                     $S \leftarrow S \cup \{(n, d^{i+3})\}$;
```

2.6.3.2. Clearances

In JP the forward and backward connection differs, since the backward connection must check cardinal-first paths. Therefore, there are four types of clearances, listed below:

(i) Forward cardinal clearances $C_f[n, d]$: These are simple clearances that reach jump points with direction d.

(ii) Forward diagonal clearances $C_f[n, d^i]$: These are recursive clearances that reach cells n' with $C_f[n', d^{i\pm1}] > 0$.

(iii) Backward diagonal clearances $C_b[n, d^j]$: These are simple clearances that reach jump points with direction $d^{j\pm3}$. Only those jump points can DFFR-reach the cell n.

(iv) Backward cardinal clearances $C_b[n, d^k]$: These are hybrid clearances that reach (1) jump points with any direction and (2) cells n' with $C_b[n', d^{k\pm1}] > 0$.

2.6.3.3. Forward connection

In Algorithm 2.6 (Harabor et al., 2019) we implement FORWARD_CONNECT. Here, we use DF-scans to systematically explore the space, but unlike SG, it is not necessary keep distance to other subgoals. The pseudo-code in Algorithm 2.6 performs the connection of the start node of a query. However, the forward connection for a jump point in S is different, performing only a subset of the scans, the ones that generate diagonal first paths, as show in Figure 2.6 (a). The connection of a start node is also shown in Figure 2.6 (b). In (Harabor et al., 2019) the authors showed the space and time complexity of running

FORWARD_CONNECT over a single jump point is $O(\min(W, H))$. However, when running this function over all jump points it is possible to limit this complexity to $O(WH)$.

Algorithm 2.6 JP forward connection from the start node located at cell n.

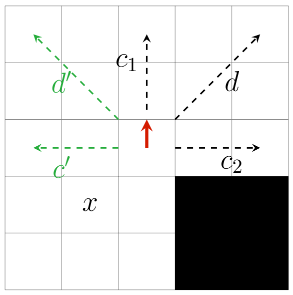
```
1: function CARDINAL_SCAN($n, i, E_n$)
2:     if $C_f[n, d^i] \geq 0$ then
3:         $E_n \leftarrow E_n \cup \{(n + C_f[n, d^i] \cdot d^i, d^i)\}$;
4: function DIAGONAL_FIRST_SCAN($n, j, E_n$)
5:     $n' \leftarrow n$;
6:     while $C_f[n, d^j] \geq 0$ do
7:         $n' \leftarrow n' + d^j \cdot C_f[n, d^j]$;
8:         CARDINAL_SCAN($n', j - 1, E_n$);
9:         CARDINAL_SCAN($n', j + 1, E_n$);
10: function FORWARD_CONNECT($n$)
11:     $E_n \leftarrow \emptyset$;
12:     for all cardinal direction superscripts $i$ do
13:         CARDINAL_SCAN($n, i, E_n$);
14:     for all diagonal direction superscripts $j$ do
15:         DIAGONAL_FIRST_SCAN($n, j, E_n$);
16:     return $E_n$;
```

2.6.3.4. Backward connection

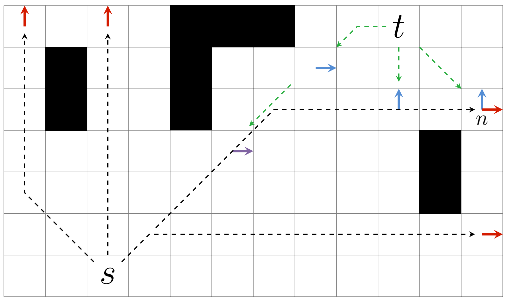
As above mentioned, backward connection must perform cardinal-first scans. A *cardinal-first* scan (CF-scan) consists of an iteration over a cardinal axis in where in each step at least one diagonal scan is performed. A *diagonal* scan consists of checking a diagonal clearance in order to perform a connection when applicable. As the authors of (Harabor et al., 2019) mentioned, CF-scans need to iterate in both cardinal and diagonal axes in order to connect all jump points that DDFFR the goal. This is shown in Figure 2.6(b). As a result, the time and space complexity of backward connection increases to $O(WH)$, which is higher than a single call to FORWARD_CONNECT.

2.6.3.5. Refine

In the refine procedure, JP must refine each edge between jump points in the corresponding freespace-diagonal-first path on G.



(a) Cardinal and DF-scans performed in forward connect.



(b) Start and target connection.

Figure 2.6. (a) Forward connect from the red jump point only calls CARDINAL_SCAN for c_1 and c_2 and DIAGONAL_FIRST_SCAN for d. Scans through green directions c' and d' are not diagonal first unless x is a blocked cell. (b) In JP the start s connects to all red nodes using cardinal or DF-scans (black lines). It does not connect to the purple node, since it is reached in a diagonal movement. However in JPD, the start would connect to the purple node and end the scan, thus not connecting to the red node in n. In JP the target t connects to blue nodes using cardinal, diagonal and CF-scans (green). Purple node is connected to the target after an iteration through the diagonal green axis. In JPD, the target do not need to iterate in diagonal directions, thus not connecting to the purple node.

2.6.4. Diagonal Merged Jump Point Graphs

In order to improve the complexity of backward connection, authors developed a variant of JP that does not need to iterate through diagonal directions. This is achieved by using additional jump points with diagonal directions along with the straight jump points. The usage of jump points with diagonal directions is explained in depth in Section 3.1.3.1. This way, a DF-scan in the forward connect procedure can stop if there is a diagonal jump point with the same direction. As a consequence, forward connections may finish earlier than in JP and backward connections now only need to perform a single diagonal scan in each visited cell in the diagonal scan. In Figure 2.6 it is possible to notice the differences between forward and backward connections of JP and JPD. The graph resulting of these modifications is called called *Diagonal Merged Jump Point Graph* **[JPD]** (Harabor et al., 2019).

Since in JPD the backward connection iterates through every cardinal direction while performing constant operations in each visited cell, its complexity is $O(W + H)$ in both time and space. The authors signaled that in the MovingAI benchmark (Sturtevant, 2012) JPD uses 26.3% and 75% less edges than JP in games and street-1024 categories respectively. Also, the connection procedure is 11% faster overall.

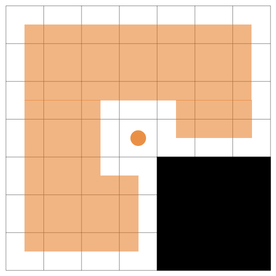
3. DIRECTED SUBGOAL GRAPHS

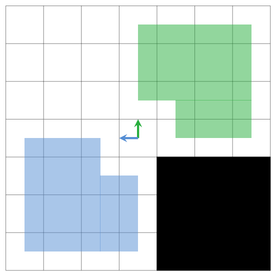
3.1. Directed Subgoal Graphs

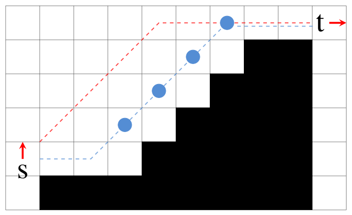
In this chapter, we introduce *Directed Subgoal Graphs* **[DSG]** which is a new subgoal graph inspired by both JP and SG. First, we provide a motivation for the key idea of DSG. Then, we proceed to the formal definition of the set of nodes called *directed subgoals* and the *direct-diagonal-first safe-freespace reachability* relation. Next, we prove that the set of *directed subgoals* forms a DFSFR-SPC on G. After that, we explain how the subgoal graph framework is implemented in DSG. Finally, we give new insights on how to improve CH performance in combination with a directed grid graph such as DSG, JP or JPD.

3.1.1. Motivation

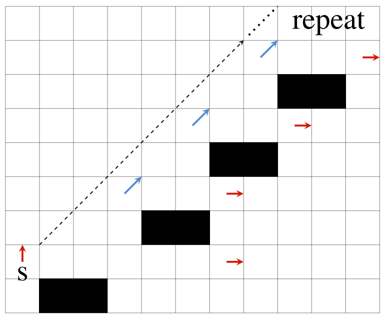
There are two properties of JP that explain its superior performance with respect to SG. The first one is that JP partitions the edge set of each corresponding subgoal in SG into two to four jump points located at the same position, as shown in Figure 3.1 (a) and (b). In case that a search needs to expand only one of those jump points, the branching factor would be reduced. Since each jump point is reached from a different direction, this condition holds. The second property is that a single JP edge may represent a path of any length in SG. An example of this is shown in Figure 3.1(c). Here, in order to go through a staircase, SG expands a node in each stair rung while JP connects directly to the end of the staircase. This has a direct impact in the depth of solutions: JP finds solutions with lower depth than SG. Although these properties present a huge potential, there are cases where JP and JPD perform poorly. Figure 1 (d) represents a floating staircase pattern that continues along the diagonal axis. In this pattern, a search execution in JP would process a number of nodes proportional to the length of the staircase, regardless of the target destination. Then, the question arises as if it is possible to use each *stair rung* as an anchor to the rest of the staircase. Thus, the number of processed nodes would be proportional to the distance to the destination. The answer to this question is explained in this chapter where we build a


(a) A single subgoal


(b) Two jump points



(c) Paths in SG and JP



(d) Staircase pattern in JP

Figure 3.1. (a) In SG, the Orange subgoal reaches any subgoal in the orange area. (b) In JP, blue and green jump points can only reach jump points in the respective blue and green area, since any other position is not diagonal-first. (c) An s-t path in SG (blue) uses four additional subgoals, while in JP the path consists of single edge (red). (d) The jump point in s connects to every red jump point in the floating staircase, therefore, when s is expanded during a search, the number of processed red nodes is proportional to the length of the staircase, regardless of the distance to the target. Using blue nodes as an anchor to the rest of the staircase could avoid this problem.

directed graph similar to JP and JPD that preserves the two properties mentioned above. Also, this graph has a more restrictive connection, i.e. edges represents paths of fewer edges on G. This connection is similar to SG and its safe-freespace-reachability.

3.1.2. Formal definition

We start by modifying the definition given by (Harabor et al., 2019) and propose the following definition for *direction extended grid graph*.

Definition 3.1. *Let $G = (V, E)$ be a grid graph. Its corresponding direction-extended grid graph G^* is defined as the tuple (V^*, E^*) where:*

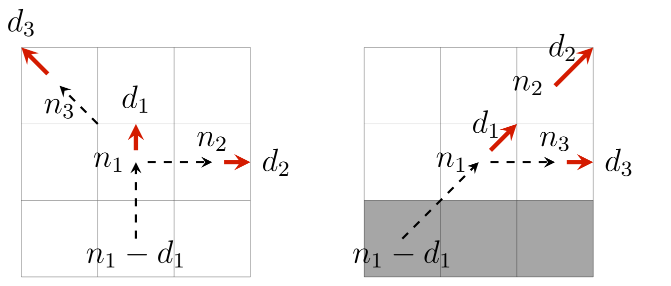
- $V^* = \{(n, d) \mid n \in V, d \in D\}$
- *For each $(n_1, n_2) \in E$ and each pair of directions d_1 and d_2 there exists $((n_1, d_1), (n_2, d_2)) \in E^*$ if and only if:*
  - (i) $n_1 + d_2 = n_2$
  - (ii) *It holds that if $(n_1 - d_1, n_1) \in E$ then $\langle n_1 - d_1, n_1, n_2\rangle$ is a shortest path; otherwise $d_1 = d_2$.*

Intuitively, as with being at node (n, d) in the direction-extended graph should be interpreted as "we are at node n *while coming* from another node in G by applying movement d". This is established by Condition 1 in Definition 3.1. Condition 2 of Definition 3.1 establishes that connections cannot be produced between nodes if such connections would allow the existence of suboptimal paths. Notice that if $n_1 - d_1$ does not exist—this could be due to the fact that n_1 is at the border of the grid or adjacent to an obstacle—we only ask that $d_1 = d_2$ to be consistent with the fact that connections in G^* reflect only shortest-path connections.

Figure 3.2 shows examples of edges and non-edges of G^* in two interesting cases.

Now we provide the definitions necessary to define our subgoal graphs. We start off with a definition for subgoal, whose location in the grid coincide with convex corners of obstacles. In other words, their location coincide with the locations of subgoals in Subgoal Graphs. Our definition is, however, slightly more involved since our nodes are direction-extended. Definitions 3.2, 3.3, and 3.4 define a subset of directed subgoals.

Definition 3.2. *A straight subgoal on a direction-extended graph G is a tuple (n, d) where n is an unblocked cell, d is a cardinal direction, $n - d$ is unblocked, and for some direction d' perpendicular to d, $n - d'$ is unblocked and $n - (d + d')$ is blocked in G.*



(a) $((n_1, d_1), (n_3, d_3))$ is an edge, but $((n_1, d_1), (n_2, d_2))$ is not.

(b) $((n_1, d_1), (n_2, d_2))$ is an edge, but $((n_1, d_1), (n_3, d_3))$ is not.

.

Figure 3.2. (a) The path $\langle n_1 - d_1, n_1, n_3\rangle$ is a shortest path but $\langle n_1 - d_1, n_1, n_2\rangle$ is not. (b) In presence of an obstacle an edge can lead to a blocked cell, but this is only allowed for edges whose $d_1 = d_2$ according to Definition 3.1

Definition 3.3. *A diagonal subgoal on a direction-extended graph G is a tuple (n, d) where d is a diagonal direction if and only if for some diagonal direction $d^k \in D$ perpendicular to d is true that $n + d^k$ is blocked and both $n + d^{k-1}$ and $n + d^{k+1}$ are unblocked in G.*

Definition 3.4. *The set of **directed subgoals** for a direction-extended graph G^* is the set of all straight and diagonal subgoals of G^*.*

Now we propose a reachability relation that holds SFR in a least restrictive fashion. This relation is used to define the subgoal reachability relation.

Definition 3.5. *A path $\langle v_1, v_2, ..., v_k\rangle$ on a direction-extended grid graph G^* with $v_i = (n_i, d_i)$ for every $i \in \{1, \ldots, k\}$, is diagonal-first (DF) if the following conditions hold:*

(i) *$\langle n_1, ..., n_k\rangle$ is a diagonal-first path on G and*

(ii) *If $(n_1 - d_1, n_1) \in E$ then $\langle n_1 - d_1, n_1, n_2, \ldots, n_k\rangle$ is a diagonal-first path on G; otherwise, $d_1 = d_2$.*

Definition 3.6. *Two nodes $v_1 = (n_1, d_1)$ and $v_2 = (n_2, d_2)$ are diagonal-first safe-freespace-reachable **[DFSFR]** if*

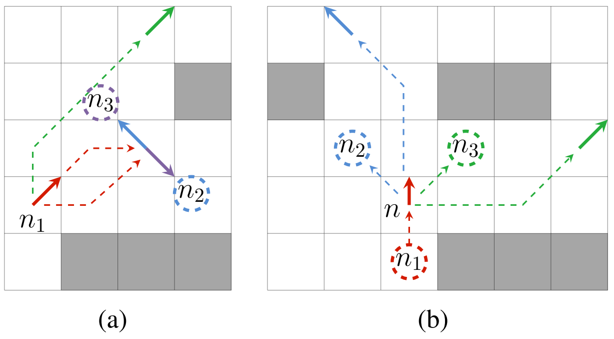


Figure 3.3. In (a) there is no path from the red subgoal to the blue and purple subgoals on G^*, since the cells associated to their incoming directions, n_2 and n_3 respectively, do not pass through any red shortest path. In (b) the path $\langle n_1, n, n_2\rangle$ is not diagonal-first, violating Condition 3 from Definition 3.6 and thus the blue subgoal is not reachable from the red subgoal . This does not happen for $\langle n_1, n, n_3\rangle$, allowing the connection to the green subgoal.

(i) (n_1, n_2) *are SFR on* G *and*

(ii) *There exists a path between* v_1 *and* v_2 *on* G^* *and*

(iii) *For the node* $v' = (n_1 + d_2, d_2)$ *the path* $\langle v_1, v'\rangle$ *is a shortest and diagonal-first path on* G^*.

In Figure 3.3 (a) we show how can we use Condition 2 to prune connections to SFR subgoals, this way allowing direct connections to subgoals that are further away in comparison with SG. Also, in Figure 3.3 (b) we show that Condition 3 can be used to block connections towards areas that does not represent a turn around the obstacle.

Definition 3.7. *Two nodes* s *and* t *are Direct-*R*-reachable if* $(s, t) \in R$ *and no shortest path passes through a subgoal* v *such that* $(s, v) \in R$ *and* $(v, t) \in R$ *with* $d_{s,v} + d_{v,t} \leq d_{s,t}$.

Definition 3.7 re-states Definition 2.2 for the case of Directed Subgoal Graphs. It is explicit in avoiding redundant edges while also is less restrictive than Definition 2.2 by allowing shortest paths containing other subgoals that cannot be used to construct an s-t path using edges defined by R. Using Definition 3.7 we define *direct-diagonal-first safe-freespace-reachability* **(DDFSFR)**.

Now, in order to prove SPC, we need the following definitions.

Definition 3.8. *Let s and t be two nodes where the freespace-diagonal-first path π is unblocked but s and t are not SFR. Let d and c be the diagonal and cardinal directions on the moves of π. The top-leftmost directed subgoal $v_L = (n, d')$ of π is the directed subgoal with the lowest i (tiebreak with lowest j) in a cell $n = s + i \cdot c + j \cdot d$ with $i, j \in \mathbb{N}_0$.*

Definition 3.9. *The parallelogram between two cells n_1 and n_2, denoted $n_1 \square n_2$ is the set of all cells (blocked or unblocked) that are in a n_1-n_2 freespace-shortest-path. For nodes of the form $v_1 = (n_1, d_1)$ and $v_2 = (n_2, d_2)$, $v_1 \square v_2$ is defined as equal to $n_1 \square n_2$.*

Lemma 3.1. *If s and t are reachable by a diagonal-first freespace path π but they are not SFR, then s DFSFR -reaches the top-leftmost directed subgoal v_L of π.*

PROOF. First, we show there exists a diagonal subgoal v^L inside $s \square t$. For this proof, we use Figure 3.4 as a reference. Given the s-t diagonal-first freespace path π is unblocked, the diagonal line that starts from s towards d inside $s \square t$ is unblocked. Also, since s and t are not SFR there is at least one obstacle in $s \square t$. We perform an iteration over an x axis with direction c and then over an y axis with direction d inside $s \square t$. Without loss of generality, let $o = s + ci + dj$ be the first obstacle that is found when scanning diagonally from s to t (that is, by creating parallel diagonals in the d direction from s moving towards t. Given the diagonal line that starts on $s + (i-1)c$ is unblocked, we know that cells $u_1 = o - c$ and $u_2 = o - c + d$ are unblocked (refer to Figure 3.4 for the location). Since the diagonal line passing through $s + (i-2)c$ is also unblocked, cell $n = o + d - 2c$ is unblocked too. Therefore a diagonal subgoal $v^L = (n, d)$ exists.
Now we show that $(s, v^L) \in DFSFR$. Since o is the first blocked cell in the iteration we know s and v^L are SFR. Also, given the s-t diagonal-first freespace path exists, we know that start direction s_t must be either d or $d-c$ and the target direction d_t must be d or c. For $d_s = d$ then for any d_t it holds that $\langle s, ((s+d_t), (d_t)\rangle$ is diagonal first. For $d_s = d-c$, a cardinal direction, there must be an obstacle in cell z that allows diagonal-first path towards t, therefore also making $\langle s, ((s+d_t), d_t)\rangle$ diagonal-first. Therefore $(s, v^L) \in DFSFR$. $\square$

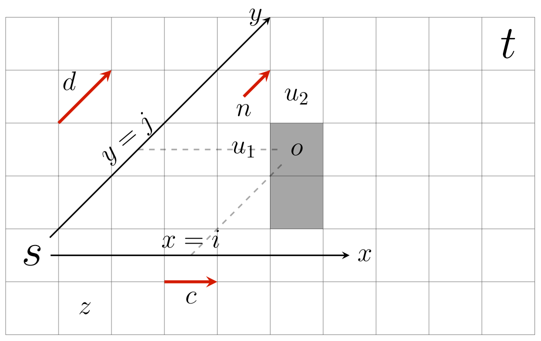


Figure 3.4. Schema of Lemma 3.1 proof.

Finally, in order to build a new subgoal graph we enunciate the R-SPC theorem.

THEOREM 3.1. *The set of directed subgoals form a DFSFR -SPC on G.*

We prove this on G instead G^* so that for two nodes s and t in G we can choose a specific direction to guarantee DFSFR . This proof uses the same idea from Theorem 2.1.

PROOF. The proof is by contradiction. Suppose s and t are not DFSFR for any direction of the start and target d_s and d_t, respectively, and no directed subgoal passes through any shortest path π. We need to show that either a shortest path passes through a directed subgoal or that s and t are DFSFR for some start goal and goal directions.

Firstly, if s and t are SFR, all freespace shortest path are unblocked on G, therefore we can choose arbitrarily d_s and d_t that make Condition 2 and 3 from Definition 3.6 true. Therefore, s and t cannot be SFR.

Let π be the diagonal-first path between s and t. As shown in Figure 2.4, we analyze the different types of turn points that π could have. If π has a card-to-card turn point, then π must pass through by a directed subgoal. If π has a card-to-diag turn point, either (1) π is not diagonal-first (contradiction) or (2) π passes through a directed subgoal. Path π cannot have diag-to-diag turn points since that would imply that π is not a shortest path. Finally, π can have diag-to-card turn points and therefore the path consists of a single turn point. For this to occur, the s-t diagonal-first freespace-path π must be unblocked. Since s and t

are not SFR, s reaches the top-leftmost directed subgoal v_L(Definition 3.8), therefore, all shortest paths from (s, d) to v_L are unblocked on G^*. Let π_1 be any of these paths. Next, the path from v_L that moves only in d has guarantee of being unblocked, at least until it reaches the cardinal section of π in a cell n_C. Let such path from v_L to v_C be π_2. Finally, the path π_3 from (n_C, c) to (t, d) is unblocked since the s-t diagonal-first freespace-path is unblocked. Therefore, the path $\pi' = \pi_1 \cdot \pi_2 \cdot \pi_3$ is a shortest path that passes through a directed subgoal v_L, leading to a contradiction. □

With Theorem 3.1 we can build a new subgoal graph according to Definition 2.4.

Definition 3.10. *$G_S = (S, E_S)$ with S as the set of directed subgoals and E_S defined by DFSFR is a subgoal graph. This subgoal graph is called Directed Subgoal Graphs **[DSG]**.*

With the incorporation of DSG to the subgoal graph framework, there are a total of 4 subgoal graphs: SG, JP, JPD and DSG. To refer any of these graphs we use subgoal graphs (lowercase) and to refer any of the directed graphs we use directed subgoal graph (lowercase).

3.1.3. Framework implementation

First we discuss the implementation of the IDENTIFY_SUBGOALS procedure of the framework. In order to commonly refer to subgoals, jump points and directed subgoals, we use the original subgoal concept.

3.1.3.1. Identify subgoals

One of the main differences between DSG and JP is that DSG uses both straight and diagonal subgoals. This idea is also present in JPD (Harabor et al., 2019) where the authors mentioned that it does not increase the total number of subgoals, since both types almost

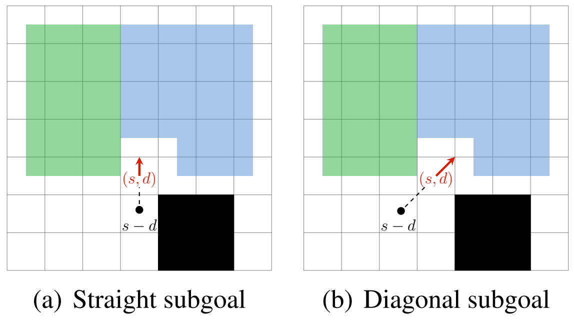


Figure 3.5. (a) Any path starting from $s - d$ to the blue area cannot be DF and any cell in the green area can be reached by a DF path. (b) Similar to (a), but paths to the blue area are also not shortest path.

always have exactly the same edges. In Figure 3.5 (a) and (b) we show that straight and diagonal subgoals reach the same area.

The algorithm to define subgoals is shown in Algorithm 3.1. Once a subgoal position is found (Lines 4-5) it adds the diagonal subgoals to the set of subgoals (Lines 7 and 10) only if they were not there before (Lines 6 and 9). Also, it adds references from straight subgoals to diagonal subgoals (Lines 8 and 11). If the set and hash are constant-time operations, the procedure is $O(1)$.

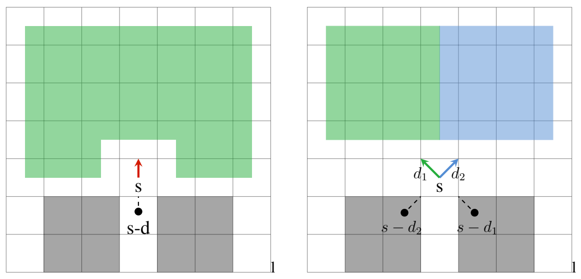
Despite the aforementioned, there are exceptions where we use straight subgoals over diagonal subgoals. When a subgoal is surrounded two convex corner cells that form a corridor, as shown in Figure 3.6, using diagonal nodes would result in multiple expansions for the same position instead of a single one. Therefore, in these cases we prefer to use the straight subgoals, and keep diagonal subgoals only as a reference to the straight ones. This modifies Algorithm 3.1, introducing new constant-time operations, but does not increase the time complexity or memory requirements.

3.1.3.2. Clearances

At a formal level, the clearances needed to detect DFSFR subgoals are as follows:

Algorithm 3.1 DSG: Identify subgoals in cell n

```
1: $R$ = an empty hash table;                                  ▷ Node references
2: function IDENTIFY_SUBGOALS($n, S, A, R$)
3:     for all diagonal directions superscripts $i$ do
4:         if $n + d^i$ is blocked on $A$ then
5:             if $n + d^{i+1}$ and $n + d^{i-1}$ are unblocked on $A$ then
6:                 if $(n, d^{i+2}) \notin S$ then
7:                     $S \leftarrow S \cup \{(n, d^{i+2})\}$;
8:                     $R[(n, d^{i+3})] \leftarrow (n, d^{i+2})$;
9:                 if $(n, d^{i-2}) \notin S$ then
10:                    $S \leftarrow S \cup \{(n, d^{i-2})\}$;
11:                    $R[(n, d^{i-3})] \leftarrow (n, d^{i-2})$;
```



(a) Single Cardinal node (b) Two Diagonal nodes

Figure 3.6. (a) The straight node allows the search to explore the green area (b) Two diagonal nodes explores half the area each one, while also sharing a column. If s is reached by the path between the obstacles, a search would need to expand both (s, d_1) and (s, d_2).

(i) Diagonal clearances $C[n, d]$: These are simple clearances that reach directed subgoals with direction d.

(ii) Cardinal clearances $C[n, d^i]$: These are hybrid clearances that reach (1) subgoals with any direction and (2) cells n' with $C[n', d^{i+1}] > 0$ or $C[n', d^{i-1}] > 0$.

In Figure 3.7 we provide an example of how can we use these clearances for detecting DFSFR subgoals. There, diagonal clearnaces can be used to detect diagonal subgoals reached by diagonal paths and cardinal clearances can be used to detect diagonal subgoal reached by CF paths and straight subgoals reached by cardinal paths.

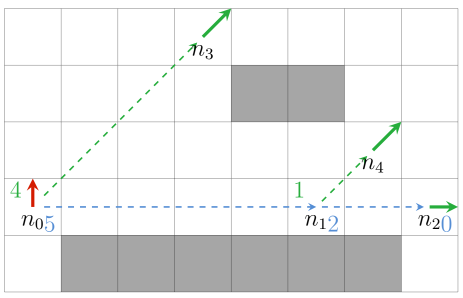


Figure 3.7. Clearances of each cell are shown in the colors of the respective direction. When detecting DFSFR from the red subgoal at n_0, the green diagonal clearance can be used to detect subgoal in n_3. Then, the blue cardinal clearance can be used to detect the cell n_1, where there is a positive diagonal clearance that can be used to detect the subgoal at n_4. Finally, the cardinal clearance also helps detecting subgoals in the same axis, such as n_2.

In Section 3.1.3.3 we explain how we can also use those clearances to preserve the direct property from Definition 3.7 and in Section 3.1.3.4 we explain how we can utilize the same clearances for the forward and backward connection.

3.1.3.3. Forward connection

The FORWARD_CONNECT procedure, is the main connection procedure which is invoked when connecting the graph (Algorithm 2.2) of a directed subgoal $v = (n, d^i)$ consists of two main function calls, (1) DIAGONAL_CONNECTION and (2) CARDINAL_FIRST_SCAN which are explained below.

The DIAGONAL_CONNECTION function explained in Algorithm 3.2 connects v to other reachable subgoals via diagonal paths. It also sets the diagonal boundaries B, which represents the maximum distance along each diagonal direction in order to keep safe and direct connections. To do so, it calls the function DIAGONAL_SCAN. This function checks a diagonal clearance to find a potentially DR subgoal while respecting and updating the boundaries. The number of calls to DIAGONAL_SCAN depends of the type of subgoal. Diagonal subgoals perform a single diagonal scan (Lines 8-9), cardinal subgoals perform two diagonal scans (Lines 11-12) and query nodes (start and goal) perform four diagonal scans (Lines 13-15).

The DIAGONAL_SCAN function performs only constant-time operations, therefore it is $O(1)$ in time and space. Given that DIAGONAL_SCAN performs a fixed number of calls to DIAGONAL_CONNECTION, its complexity is also $O(1)$.

Algorithm 3.2 Diagonal connection and diagonal scan.
In red changes needed for backward connection.

```
 1: function DIAGONAL_SCAN($n, i, b$)
 2:     $clr \leftarrow C[n, d^i]$;
 3:     if $clr > 0$ and $clr < b$ then
 4:         $E_v \leftarrow E_v \cup \{(n + d^i \cdot clr, d^{i+4})\}$;
 5:         return $clr+1$;
 6:     return $b$;
 7: function DIAGONAL_CONNECTION($n, i, B$)
 8:     if IS_DIAGONAL($d^i$) then
 9:         $B[i] \leftarrow$DIAGONAL_SCAN($n, i, \infty$);
10:     else if IS_CARDINAL($d^i$) then
11:         $B[i-1] \leftarrow$ DIAGONAL_SCAN($n, i-1, \infty$);
12:         $B[i+1] \leftarrow$DIAGONAL_SCAN($n, i+1, \infty$);
13:     else                                                    ▷ Query node
14:         for all diagonal direction superscripts $i$ do
15:             $B[i] \leftarrow$ DIAGONAL_SCAN($n, i, \infty$);
```

The CARDINAL_FIRST_SCAN function explained in Algorithm 3.3 is where most of the connections occur. It iterates through a cardinal axis and in every step it checks if a diagonal connection is feasible, what we call a cardinal-first scan (CF scan). We also allow each diagonal scan to run independently and use the boolean parameters do_CCW and do_CW to determine if the corresponding diagonal scan must be performed. When one of these parameters is initially *false*, we say that it is a half CF scan. An example of CCW and CW scans is shown in Figure 3.8 (a).

In detail, this function first sets the boundaries to the CW and CCW diagonal scans b^- and b^+ according to DIAGONAL_CONNECTION (Lines 2 and 3). Then, the main loop starts (Line 7). This loops iterates through the cardinal direction d^i where each step is determined by the cardinal clearance. As shown in Figure 3.7, it can be used for the connection to subgoals reached by cardinal paths (Lines 9-11) and cardinal-first paths (Lines 16-19).

Additionally, if there is a subgoal with direction $d^{i\pm2}$ in the cardinal axis, it reveals the existence of an obstacle and thus the corresponding diagonal scan must halt (Lines 12-15). Given that this loop perform two types of connections, the second condition to break the loop consists of (1) there are no more CF scans to perform and (2) a successful cardinal connection has already been performed (Line 7).

The cost of calling CARDINAL_FIRST_SCAN consists of a for loop through a cardinal axis of the grid in which calls to IS_SUBGOAL, DIAGONAL_SCAN and other constant-time operations are performed. The auxiliary function IS_SUBGOAL(n, d) returns true if there is a directed subgoal with that position and direction. This function can run in constant time with an array or a hash table. Given that DIAGONAL_SCAN is also $O(1)$, the time and space complexity of CARDINAL_FIRST_SCAN is $O(W + H)$ in the worst case.

Algorithm 3.3 DSG forward cardinal first scan

```
1:  function CARDINAL_FIRST_SCAN($n, i$, do_CCW, do_CW)
2:      $b^- \leftarrow B[i-1]$;                                   ▷ CW(-) boundary
3:      $b^+ \leftarrow B[i+1]$;                                   ▷ CCW(+) boundary
4:      $clr \leftarrow C[n, d^i]$;
5:      $n' \leftarrow n$;
6:      connected_in_cardinal ← false;
7:      while ($clr > 0$ and (do_CCW or do_CW or not connected_in_cardinal)) do
8:          $n' \leftarrow n' + clr \cdot d^i$;
9:          if IS_SUBGOAL($(n', d)$) and not connected_in_cardinal then
10:             $E_v \leftarrow E_v \cup \{(n', d)\}$;
11:             connected_in_cardinal ← true;
12:         if IS_SUBGOAL($(n', d^{i-2})$) then
13:             do_CCW ← false;                                    ▷ Halt CCW scan
14:         if IS_SUBGOAL($(n', d^{i+2})$) then
15:             do_CW ← false;                                     ▷ Halt CW scan
16:         if do_CCW then
17:             $b^+ \leftarrow$DIAGONAL_SCAN($n', i+1, b^+$);     ▷ CCW scan
18:         if do_CW then
19:             $b^- \leftarrow$DIAGONAL_SCAN($n', i-1, b^-$);     ▷ CW scan
20:         $clr \leftarrow C[n', d^i]$;
```

The FORWARD_CONNECT function is explained in Algorithm 3.4. The number of calls and the parameters used in CARDINAL_FIRST_SCAN depends of the type of subgoal being connected. Diagonal subgoals perform two CF-scans halves (Lines 5-7), cardinal subgoals performs one complete CF-scan and CF-scans two halves (Lines 8-11) and query nodes perform all CF-scans (Lines 12-14). An application of this algorithm is illustrated at Figure 3.8 (b).

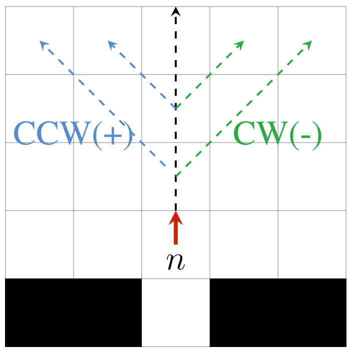
The complexity of the FORWARD CONNECT algorithm consist on (1) Diagonal connection which is $O(1)$, (2) Up to four cardinal scans which are $O(W + H)$. Thus, FORWARD CONNECT is $O(W + H)$ in time and space.

Algorithm 3.4 DSG forward connection from subgoal $v = (n, d^i)$.

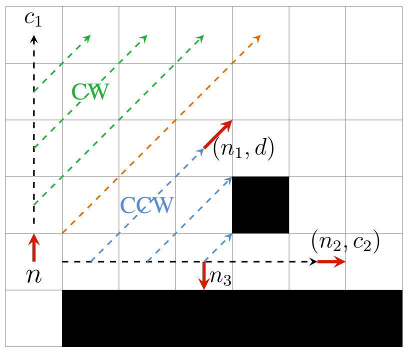
```
 1: function FORWARD_CONNECT($(n, i)$)
 2:     $E_v \leftarrow \emptyset$;
 3:     $B \leftarrow$ array of size 8;                         ▷ Diagonal boundaries
 4:     DIAGONAL_CONNECTION$(n, i)$;
 5:     if IS_DIAGONAL$(d^i)$ then
 6:         CARDINAL_FIRST_SCAN$(n, i - 1,$ true, false$)$;
 7:         CARDINAL_FIRST_SCAN$(n, i + 1,$ false, true$)$;
 8:     else if IS_CARDINAL$(d^i)$ then
 9:         CARDINAL_FIRST_SCAN$(n, i,$ true, true$)$
10:         CARDINAL_FIRST_SCAN$(n, i - 2,$ true, false$)$;
11:         CARDINAL_FIRST_SCAN$(n, i + 2,$ false, true$)$;
12:     else                                                    ▷ Query node
13:         for all cardinal directions $i$ do
14:             CARDINAL_FIRST_SCAN$(n, i,$ true, true$)$;
15:     return $E_v$;
```

3.1.3.4. Backward connection

The backward connection answers the question of which nodes DDFSFR-reach the goal. In JP and JPD, since the only valid path between two nodes is the DF path, the goal must be connected using a CF-scan, needing additional reversed clearances for this purpose. However, in DSG the goal connection can maintain the forward connection schema using CF-scans too since any path is valid between two DDFSFR nodes. Therefore this procedure does not need to use additional clearances, halving the memory used for this purpose. Finally, there are three differences that are discussed below.



(a) A DSG cardinal-first scan



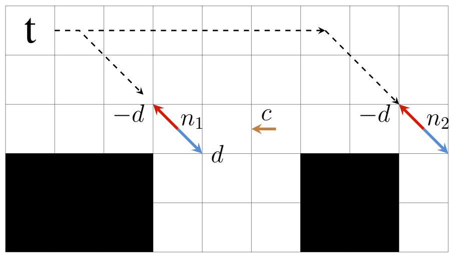
(b) Forward connect over a diagonal subgoal

Figure 3.8. (a) The CF-scan from n towards the blue lines is CCW and towards the green lines is CW (b) Connection of a directed subgoals is performed in two halves CF-scan. Here, we can notice that the CCW diagonal scan along d ends in n_3, given that $(n_3, -c_1)$ reveals the existence of an obstacle towards d. However, the cardinal connection continues up to n_2.

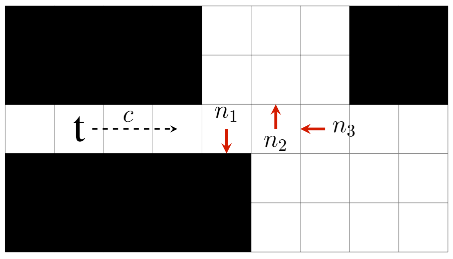
(i) As show in fig. 3.9 (a), a CF scan from t in directions c and d would detect subgoals of the form (n, d) pointing outwards the target. However, for Definition 3.3 it is guaranteed that the subgoal $(n, -d)$ exists, which points towards the target. Therefore, this subgoal can DDFSFR-reach the target. Thus, in Algorithm 3.2, the only modification needed to fulfill this behaviour is changing d^i to d^{i+4} in Line 4.

(ii) In FORWARD_CONNECT if there are multiple DFSFR subgoals in a line with a parallel direction to the cardinal scan direction, only the first reached subgoal is direct, given that it R-reaches the other subgoals in the same cardinal line. However this does not happen in backward connection. In Figure 3.9 (a) we can see that the brown subgoal does not have a diagonal first movement towards the target, therefore the direct condition between n_2 and t holds. Considering this, in order connect to all DDFSFR-reachable nodes we must change Line 5 in Algorithm 3.2 by widening the margin by 1.

(iii) The aspect in which the forward and backward connections differ the most is in the connection to nodes reached by cardinal paths. In the target connection, t can



(a) Target CF-scan



(b) Target cardinal connection

Figure 3.9. (a) The backward connection from t can use CF-scans from FORWARD_CONNECT to find (n_1, d) and (n_2, d). However, the reached subgoals point outwards t. Therefore, the backward connection must connect subgoals with direction -d. Also, in FORWARD_CONNECT (n_2, d) would not be direct to t, since (n_1, d) DDFSFR-reaches (n_2, d), breaking Definition 3.7. However, in backward connection this no longer happens since any subgoal DFSFR-reached from $(n_2$,-$d)$ along c does not have a diagonal-first path towards t, as the brown node. Thus, the target is direct from $(n_2$,-$d)$. (b) Up to three nodes can directly reach the target t along c, one at the left, one at the right and one at the opposite direction with respect to the scan direction c.

be connected up to three different nodes in the same line, as shown in Figure 3.9. We can also notice that t no longer connects to (n_1, c).

The outline of the backward CF-scan is: Connect t to the first node in each perpendicular direction $c' \perp c$, which also triggers the halting of the corresponding CF-scan. The entire CF-scan finishes when it finds and connects to a node in the direction $-c$. Since the backward connection is a modified version of Algorithm 3.4, Algorithm 3.2 and Algorithm 3.3 that only adds constant-time operations, the time and space complexity remains at $O(W + H)$.

3.1.3.5. Refine

Since every pair of DDFSFR subgoals are also SFR, we can use any freespace shortest path for the refine procedure. For simplicity, we refine using DF paths.

3.2. Contraction Hierarchies and the subgoal graph framework

Since there are different implementations for the subgoal graph framework and its corresponding CH graphs, we use *CH-subgoal-graph* **[CH-sg]** to refer to any subgoal graph. If the graph is also directed, as in the case of JP, JPD and DSG, we use *CH-directed-subgoal-graph* **[CH-dsg]**.

3.2.1. Overview

- **Preprocessing phase**: It consists of building the subgoal graph without any alteration and then running CH over it, generating the CH-sg.
- **Query phase**: In the connect procedure, it assumes that s and t are the in the lowest hierarchy level, therefore, all the connection edges needed in a subgoal graph are needed in a CH-sg too. (2) In the search phase, one can use any bidirectional search algorithm, although Bidirectional Dijkstra is the most common. However, in (Uras, 2019) the authors used Bidirectional A* instead of Bidirectional Dijkstra, since it reported faster searches. (3) In the refine phase, we group the unpacking procedure of CH and the refine procedure of the subgoal graph.

3.3. Improvements to the subgoal graph framework

In this section we provide some optimizations to the subgoal graph framework and its interactions with CH. For that purpose, we use the word *subgoal* to refer to a node v in the set of subgoals s and *directed subgoals* to refer to a subgoal with direction.

3.3.1. Avoidance

In (Harabor et al., 2019) the authors developed a method called *avoidance tables*. This optimization was developed in order to decrease the number of connections a query point performs to nodes that can be considered as useless during a search, i.e. nodes that have zero outgoing or incoming edges.

We modify this method by defining nodes without incoming edges as *backward avoidable* and nodes without outgoing edges are *forward avoidable*. An avoidable node can be skipped either in the connection or the search, but the avoidable state must be updated before in the connection of the query point that adds edges to the opposite graph. Once a subgoal graph is built, one can determine the *avoidance* status of every subgoal in the forward and backward graph. Then the *connection avoidance* and the *search avoidance* can be performed as follows:

- Connection avoidance: If a subgoal has no incoming edges, it cannot be reached by another subgoal. Therefore, when solving a query, if the backward connection reach this subgoal, the connection would be meaningless. The only exception is when the start also connects to this subgoal. Thus the connection avoidance consists of two steps: (1) when connecting the start, mark backward avoidable nodes as not avoidable and (2) when connecting the target, skip connection to backward avoidable nodes. This procedure can also be used in forward connection, but requires the target to be inserted previously, leading to a small overhead.
- Search avoidance: If a subgoal different from the target has no outgoing edges, adding it to the open is unnecessary. The procedure consists of the following steps: (1) when connecting the target, mark each reached node as not forward avoidable. (2) when expanding a node, if a successor is forward avoidable, then it can be ignored. For bidirectional search this procedure can also be used in the backward search.

3.3.2. Reducing redundant edges in CH-dsg

In (Uras, 2019) the authors explained most of the drawbacks of this process for CH-dsgs. One of these is explained as follows. The contraction process of a node v verifies if without v, the remaining graph preserves the shortest distance between two incoming and outgoing neighbours of v, w and u. However, in CH-dsgs there are multiple nodes in

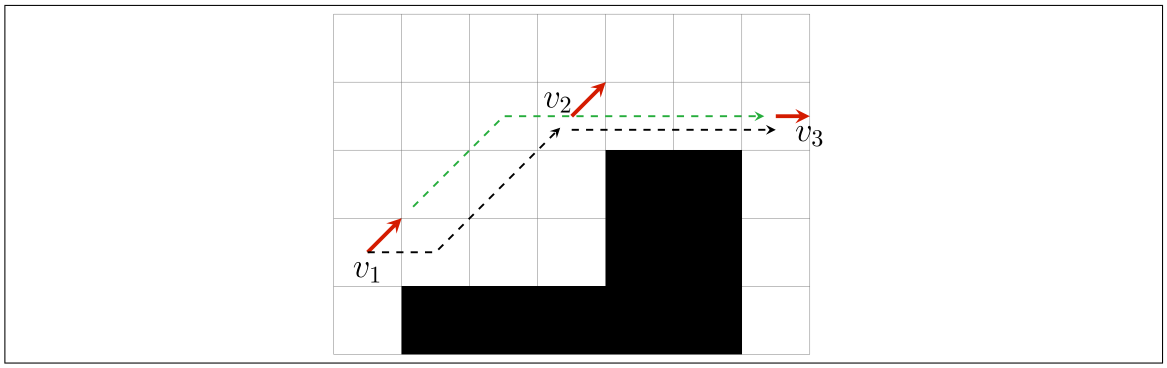


Figure 3.10. The green shortcut between v_1 and v_3 that passes through v_2 can be unpacked without using the black edges if we use freespace-DF-reachable shortcuts and refine it into the freespace-DF path.

each cell, which results in that the distance between two cells can be preserved even when there is only a couple of shortest path between any two nodes in those cells. Thus, the w-u distance may be preserved even if the contraction of v does not detect it, since other nodes in w and u positions may still have a shortest path. When this happens, a redundant shortcut is added to the graph.

Thus, CH adds many redundant shortcuts. The authors reported up to 25.6% of redundant edges in a random map. In order to detect if a shortcut is redundant, one must compare its cost with the distance between the cells d_{n_2,n_3}. For that purpose, the distance matrix between every subgoal position M must be constructed. The authors proposed an optimal and a suboptimal method to deal with this problem. The optimal method consists of performing a Dijkstra search from every subgoal position s and use as start nodes every subgoal (s, d) with $d \in D$. However, the authors reported it increased the preprocessing time of CH in up to three times. Therefore, they proposed a suboptimal approach in which a Dijkstra search is run from only the subgoals that already exists on G_S. However, this search is not complete, since it skips the *try direct path* and *Connect* phases of the subgoal graph query phase. Given this, after this procedure they reported that 15.9% of the edges were redundant.

Redundant edges not only translate to a *larger* graph, they also can lead to an incorrect calculation of the elimination priority of a node and therefore, deteriorating the node ordering and the query times. In this section we further provide two additional approaches to reduce the redundant edges problem:

(i) In grid graphs the distance between a pair of nodes is always symmetrical. Thus, if $M'[s_1, s_2] < M'[s_2, s_1]$, we can make $M'[s_2, s_1] = M'[s_1, s_2]$

(ii) We can detect all pair of nodes (s_1, s_2) that are reachable by a freespace diagonal-first or cardinal-first path and set their distance as $M'[s_1, s_2] = \text{octile\_distance}(s_1, s_2)$. This process can be sped up using clearances that do not need to be stored afterwards.

3.3.3. Unpacking freespace-R-reachable shortcuts

Finally, we present an optimization in the unpacking procedure that was also discussed in (Uras & Koenig, 2018). There the authors mentioned that, for CH-SG, is possible to avoid saving the unpacking information of a shortcut. This occurs when a shortcut is added between two subgoals s_1 and s_2 that satisfies a given freespace reachability relation. For example, if (s_1, s_2) are freespace-diagonal-first reachable ($\mathcal{F}$DFR) we can avoid saving the unpacking information and refine the edge directly into the freespace path, thus improving the refine process. However, this idea is not bound only to SG, therefore it can be used in any CH subgoal graph. When we use this optimization we say that we are *using* freespace-R reachable shortcuts ($R_{\mathcal{F}}$R) shortcuts. The benefit of using this optimization is that in the refine procedure, we can skip the unpacking procedure for $R_{\mathcal{F}}$R shortcuts following the $R_{\mathcal{F}}$ reachability relation to produce a path on G, as shown in Figure 3.10.

4. EXPERIMENTAL EVALUATION

4.1. Experimental setup

In this section we evaluate the time and memory performance of the algorithms we have discussed in this thesis. For that porpuse, in Section 4.1.1 we introduce the benchmark that is used to evaluate and compare all these algorithms. Then, in Section 4.1.2 we explain the validation process to compare JP and JPD with DSG. Next, in Section 4.1.3 we explain implementation details of the search algorithms, subgoal graphs and CH.

4.1.1. Benchmarks

We use a single set of benchmarks in our experiment, the *MovingAI benchmarks* (Sturtevant, 2012), which was created to allow researchers compare pathfinding algorithms. This benchmark consists of several groups of maps, where we can distinguish 3 different sources: (1) commercial video games (2) discretized real life environments and (3) autogenerated maps. We follow the categorization used in (Harabor et al., 2019) where there are 5 main categories and a total of 26 subcategories, detailed below:

- (i) Games: Maps from different video games.
    - (a) **bg**: 120 maps from *Baldur's Gate II*.
    - (b) **bg-512**: 75 maps from *Baldur's Gate II*, scaled to 512x512.
    - (c) **dao**: 156 maps from *Dragon Age: Origins*.
    - (d) **da2**: 67 maps from *Dragon Age 2*.
    - (e) **sc1**: 75 maps from *Starcraft I*.
    - (f) **wc3-512**: 36 maps from *Warcraft III*, scaled to 512x512.
- (ii) Mazes: Set of maze maps with 512x512 resolution. The type of maze is determined by the corridor size, that can be 1, 2, 4, 8, 16, or 32. There are 10 maps for each corridor size, leading to a total of 60 maps. The subcategories are defined by the corridor size X, called *maze-X*.

(iii) Random: Set of maps with random obstacles and 512x512 resolution. Each cell has a percentage of being blocked that goes from 10 to 40 in steps of 5. This percentage X determines the subcategory, called *random-X*. There are 10 maps in each subcategory for a total of 70 maps.

(iv) Rooms: Set of room maps with 512x512 resolution. Each map is subdivided in square rooms all with the same size. Then, doors between rooms are added with a probability of 0.8. Possible rooms sizes are $8 \times 8, 16 \times 16, 32 \times 32$ and 64×64. Subcategories are determined by the room side size X, named $room$-X.

(v) Streets: Maps are a discretization of 30 different real life environments. The discretization is performed in three resolutions: 256x256, 512x512 and 1024x1024, resulting in a total of 90 maps. We have one subcategory for resolution called *street-X*.

The instances or *queries* of each map are generated randomly according to the following criteria. First, multiple problems between random points are generated. Then each problem with a shortest path of cost l belongs to a bucket $\lfloor l/4 \rfloor$. A maximum of 10 problems are selected from each bucket. The number of buckets is determined by the largest shortest path. Therefore, maps with larger shortest paths have more instances, as in the case of mazes. In the Table 4.1 we show a summary of the MovingAI benchmark, where the number of maps, instances, graph size are shown. We also show statistics for an A* run over these problems, capturing the average path length, the number of expansions that A* performs and its runtime.

We observe that solutions for the mazes benchmark are the longest paths, whereas street-1024 has the largest graphs both in number of vertices and number of edges. In games we observe that Starcraft and Warcraft III are the games with largest graphs, whereas Starcraft and Dragon Age: Origins are the games with the largest shortest paths.

Table 4.1. Statistics of solving all instances from MovingAI benchmark using A*.

	Map	Graph sizes		Instances		Sol.	N°	Time
	Count	$\mid V \mid$	$\mid E \mid$	Avg.	Total	Cost	Exp.	(μs)
all	789	127 123	916 054	2177	1 717 640	1120	56 202	31 331
games-all	529	64 861	499 013	1234	653 050	430	25 475	16 272
mazes-all	60	207 941	1 338 316	10 450	627 000	2265	101 435	51 017
random-all	70	185 864	937 004	2225	155 750	482	28 046	16 786
rooms-all	40	232 785	1 691 743	2109	84 350	422	40 454	26 669
streets-all	90	346 561	2 724 776	2194	197 490	567	43 131	32 091
bg	120	4507	32 180	340	40 780	142	2676	1460
bg-512	75	73 930	574 510	1635	122 600	360	12 829	7905
dao	156	21 322	159 548	998	155 620	418	13 256	7926
da2	67	15 911	117 640	1003	67 200	280	6010	3604
sc1	75	263 782	2 040 510	2819	211 390	612	55 092	35 769
wc3-512	36	112 488	867 184	1541	55 460	318	15 182	10 116
maze-1	10	131 071	262 140	14 549	145 490	2986	63 662	23 690
maze-2	10	174 517	870 383	11 911	119 110	2413	85 014	39 877
maze-4	10	209 268	1 356 891	10 422	104 220	2150	105 629	54 103
maze-8	10	232 928	1 688 141	10 331	103 310	2133	125 164	66 053
maze-16	10	246 042	1 871 727	9121	91 210	1877	133 310	73 419
maze-32	10	253 819	1 980 615	6366	63 660	1300	127 443	72 770
random-10	10	235 903	1 533 055	1797	17 970	359	11 258	8685
random-15	10	222 689	1 301 321	1861	18 610	372	17 341	12 400
random-20	10	209 255	1 097 499	1915	19 150	383	22 903	15 443
random-25	10	195 315	918 694	1990	19 900	398	27 775	17 910
random-30	10	180 209	760 078	2078	20 780	415	31 805	20 238
random-35	10	161 313	613 313	2297	22 970	459	36 772	21 418
random-40	10	96 365	335 069	3637	36 370	751	37 015	18 229
room-8	10	206 792	1 301 001	2098	20 980	419	36 905	24 064
room-16	10	231 263	1 663 793	2037	20 370	407	35 885	23 982
room-32	10	243 733	1 855 004	2097	20 970	419	41 255	27 485
room-64	10	249 353	1 947 173	2203	22 030	441	47 296	30 857
street-256	30	48 012	363 862	934	28 020	187	4440	2969
street-512	30	196 602	1 531 900	1876	56 270	376	16 899	11 797
street-1024	30	795 069	6 278 567	3773	113 200	755	65 747	49 387

4.1.2. Validation of JP and JPD

In order to perform a fair comparison between DSG, JP and JPD, we developed our own implementation of both JP and JPD. The differences in data structures, search algorithms, datatypes, and other micro optimization between our DSG implementation and the original JP and JPD implementation could lead to differences in execution times that do not reflect the properties of these algorithms, leading to wrong conclusions. Thus, we implemented a framework that can run all these algorithms: A*, SG, DSG, JP and JPD and the CH combinations (except for A*). This common framework treats all subgoal graphs as equivalent, where the connect procedure is specific for each, but other aspects are equivalent. In order to validate our framework we tested the graph size of JP and JPD in several maps, achieving a perfect equality. We also tested the target and goal connections for these maps, achieving a perfect equality too. With respect to the search performance, we compared the number of expansions of A* for these maps, achieving a difference of around $\pm 1\%$, mainly explained by the priority queue with double priority.

For CH, as we do not share the same implementation, we tested the speedup against A* in the grid graphs, achieving speedups of the same order of magnitude.

The implementation of all these algorithms can be found at `https://gitlab.com/bmarinb/directed-subgoal-graphs`.

4.1.3. Implementation details

- **A*:** We implemented A* using *octile distance* as the heuristic. The priority queue is a Binary Heap with decrease key operation. This queue also uses a second criterion, the h value, which is used to break ties. We preallocate all the memory needed in the execution: g, parents, heap items and indices.
- **Second priority term:** We use a new technique based on the theoretical maximum values h_{max}, g_{max}, and f_{max} in grid maps, described as follows: For a map of size $W \times H$ we have $g_{max} \leq W \cdot H$. Without loss of generality, lets assume $W \geq H$. Then $h_{max} = H(\sqrt{2} - 1) + W$ and $f_{max} = g_{max} + h_{max}$. In this scenario, the number of bits needed to store the integer part of h_{max} and f_{max} are $\lceil log_2(h_{max}) \rceil$ and $\lceil log_2(f_{max}) \rceil$ respectively. For a 1024×1024 resolution these amounts are 21 and 11 bits - summing up 32 bits. Using a 64 bit unsigned integer, there are left 32 bits for the decimal part of both terms. This is represented in Figure 4.1. With this approach it is possible to compare both g and h

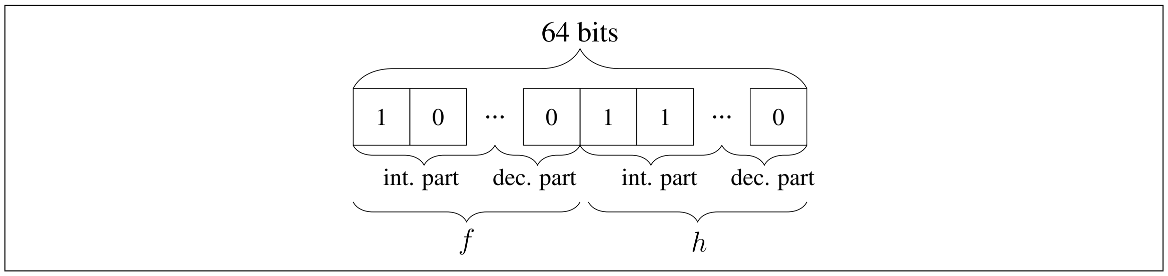


Figure 4.1. Bits schema for storing f and h in a single 64-bit integer.

using a single comparison at the cost of the computation of this priority term, that can be done with multiplication, bit shifting and addition.

- **Map representation:** We use a flattened array of size $W \times H$. Therefore, each cell (i, j) can be represented by a 4 bytes integer using position $p = i + jW$ in the array. For traveling across the map, we precompute the distance in the flattened array of moving from one cell s to other cell $s + d$ for each $d \in D$.
- **Grid graph:** Each unblocked cell is a node and its edges consists of a single byte where the i-th bit signals the existence of an edge to the direction d^i.
- **Subgoal graph framework:**
  - **Subgoal information:** For each subgoal, we save its position and direction in the flattened array p and its direction d in a single 4 byte integer as $p \times |D| + d$. We also save all reference subgoals that refer to (p, d) in a single byte, using each bit for each possible direction. Therefore, we use 5 bytes for directed subgoals and 4 bytes for subgoals.
  - **Edges:** We save edges ordered by source node. Since the cost is always the octile distance, we only keep the target in a 4 byte integer.
  - **Subgoal edges:** To identify each subgoal edges, we use a 4 byte integer for saving the first edge in the ordered array and a 2 bytes integer to save the number of edges it has. Therefore, we use a total of 6 bytes.
  - **Node detection:** We use a two-part approach that consists of a matrix that keeps the presence of different subgoal directions in a given cell. This can be saved in 8 bits (1 byte) leading to WH bytes. Then, once the existence of

a node with a given direction is determined, we use a hash table that maps the flattened tuple (position, direction_superscript) into a node identifier.

  - **Grouping subgoals**: In directed subgoal graphs, we use an optimization from (Uras, 2019) where all directed subgoals in the same cell are *grouped* into a single node in the search algorithm. Therefore we need to preallocate memory (Open, g and f) only for the number of different cells of directed subgoals, equal to the number of subgoals in SG.
  - **Removing redundant edges:** In JP and JPD the preprocessing phase generates a graph with redundant edges. In this context, a redundant edge is an edge that can be represented by a path of the same cost. In order to reduce the graph size, it is possible to delete this edges by running a modified Dijkstra from each jump point v. This search differs from the traditional Dijkstra in that it updates the parent of a node in the Open anytime it is reached by a new path with the same cost. This way, outgoing neighbours that have a parent different from v represent redundant edges that can be deleted from the graph.

- **Contraction Hierarchies over the subgoal graph framework**: The forward and backward graphs contain the information to identify the edges that correspond to each node, which are the first edge (4 bytes) and the number of edges (2 bytes). In SG the forward and backward graphs point to the same structure, while in directed subgoal graphs (DSG, JP and JPD) the forward and backward graphs are materialized separately. Each edge is saved only in one node, the one with lower hierarchy. Each edge has 5 attributes: source and target nodes, cost and unpacking information. Each attribute use 4 bytes, for a total of 20 bytes per edge. All edges are saved in the same vector. With respect to the parameters used to define the priority of a node in the contraction process are shown in Table 4.2. These values were obtained from a greedy exploration based on (Geisberger et al., 2008) as start point. For the refinement process, we use freespace-cardinal-first and freespace-diagonal-first reachable shortcuts. In order to detect if a given

Table 4.2. CH priority term weights. E = Edge Difference weight, D = Contracted Neighbours weight, S = Search Space weight and L = Search space limit.

E	D	S	L
120	120	0.5	1000

freespace reachability relation holds, we use the same clearance mentioned in Section 3.3.

4.2. Results and discussion

In this section we present and discuss the results of building all subgoal graphs over each map in the MovingAI benchmark and solving all the corresponding problems in each map, excluding the *random* due to JP-CH, JPD-CH and DSG-CH preprocessing times and *street-1024* due to high SG-CH preprocessing times. In these benchmarks we have partially lead to the same conclusions present of (Harabor et al., 2019): CH-SG and CH-JPD overperforms all other subgoal graphs in *random* and *street-1024* benchmarks respectively.

In each subsection, we briefly describe the statistics captured in the experiment and then proceed to comment the results. For statistics that are relative to the preprocessing stage, the averaging is performed weighting each category by the number of maps in it. For statistics that are relative to the query phase, the averaging is performed weighting each category by the number of problems in it. Given that maze maps have the highest number of problems per map, they tend to be over-represented in the *all* benchmark. All the query times are measured in micro seconds (μs) and the preprocessing times are measured in seconds (s).

4.2.1. Standalone subgoal graph framework

First we test the standalone subgoal graph framework, i.e, without CH. We divided the from the preprocessing and query phases.

4.2.1.1. Preprocessing phase

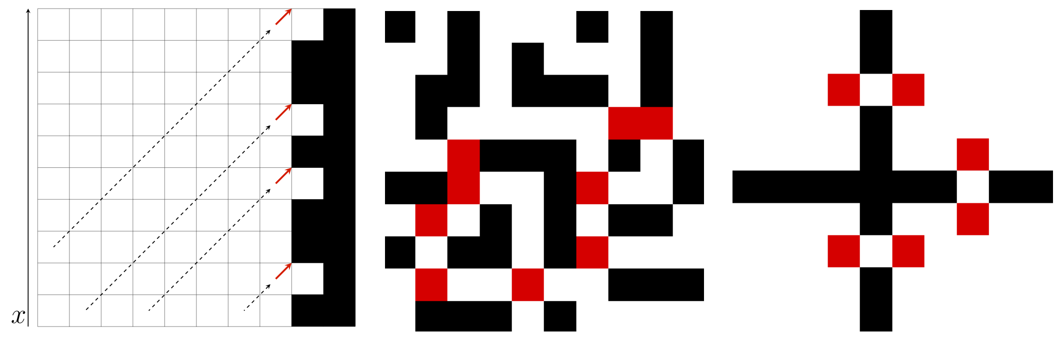
In the preprocessing phase we capture the following metrics that describe the graph size:

- $|V|$: The number of subgoals in the subgoal graph.
- $|E|$: The number of edges in the subgoal graph.

In Table 4.3 we see that with respect to the number of subgoals, the minimum number is always reached by SG since it has at most one subgoal per cell. However, with respect to the other subgoal graphs, we observe that DSG and JPD have lower number of subgoals than JP. In Section 3.1.3.1 it is explained a case where it is convenient to use straight subgoals over diagonal ones, generating one subgoal less. In the same way, there is a case when using diagonal subgoals is better that straight ones, generating two subgoal less. This happens when a subgoal is surrounded by two convex corner cells that are in opposite directions. Since JP do not make use of this optimization it has more subgoals. With respect to the number of edges, JPD takes the lead for all benchmarks except for maze-1 and rooms. The dominance of JPD can be explained by the following factors:

- The DF-scans performed by forward connect in JPD end when a jump point is found in the diagonal scan, as shown in Figure 2.6. Unlike this, JP DF-scans do not end until an obstacle is found. Therefore, JPD scans end earlier than JP scans, resulting in a lower number of edges.
- With respect to DSG, this happens mainly due to wall roughness. In a rough wall, as shown in Figure 4.2 (a), JPD ignores most jump points that point towards a rough wall, unlike DSG in which these subgoals are needed to reach the concave corners of the rough wall.

Despite the aforementioned, DSG is positioned in the 2nd place, having 17% less edges than JP in games benchmark. This is explained due to having a more restrictive connection. The only benchmarks where JPD does not leads is in maze-1 and rooms, where SG



(a) JPD and DSG in rough walls (b) maze-1 (c) rooms

Figure 4.2. (a) In JPD every red jump point can only be reached by another jump point in the marked diagonal line. In contrast, in DSG, every subgoal with a lower x-coordinate than a red subgoal can DFSFR -reach it (not necessarily directly). (b) and (c) Red positions represent unblocked cells with more than two directed subgoals. This is highly common in maze-1 and rooms benchmarks.

does. This can be explained by the number of nodes. DSG and JPD have $\times 2.01$ more nodes than SG in games benchmark, where SG has more edges overall. This is the expected result since a single convex corner cell generates one subgoal and two directed subgoals or jump points. However, in maze-1 and rooms DSG and JPD have $\times 2.34$ and $\times 2.77$ more nodes respectively. Having more than two subgoals in the same cell only can happen when a cell is surrounded by more than a convex cell. In these benchmarks, corridors and doors have a width of one cell which propitiates this condition, as shown Figure 4.2 (b) and (c). Since DSG and JPD have proportionally more nodes in these benchmarks, it also results in an increased number of edges.

4.2.1.2. Query phase

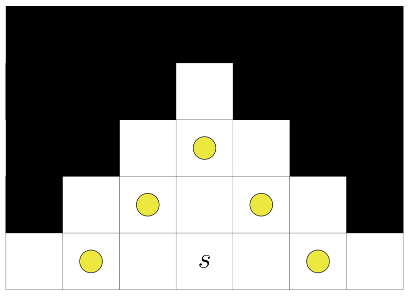
In the query phase, the time consumed for solving a path planning problem is determined by the *Connect-Search-Refine* (CSR) procedure. Besides this, there are other time consuming tasks stages such as pre-allocating memory, deleting query points s and t from the subgoal graph graph and resetting avoidance status. However, given this tasks can be performed anytime before and after the query, we do not include them in this results. In

Table 4.3. Subgoal graphs sizes in the MovingAI benchmark. All categories are included except *random* and *street-1024*.

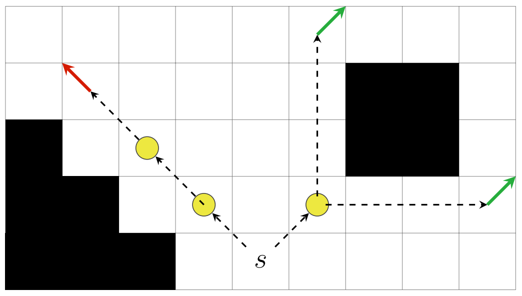
	$\mid V \mid$				$\mid E \mid$			
	SG	JP	JPD	DSG	SG	JP	JPD	DSG
all	**2593**	5596	5579	5579	25141	12882	**10756**	11361
games-all	**1425**	2887	2874	2874	21521	10017	**7924**	8304
mazes-all	**11557**	25164	25164	25164	**27447**	29853	29768	30195
rooms-all	**4270**	11908	11848	11848	**13797**	18128	17940	18029
streets-all	**2806**	5698	5668	5668	62310	17672	**11921**	15026
bg	**449**	922	909	909	4307	1852	**1305**	1448
bg-512	**666**	1334	1334	1334	8164	2573	**2124**	2961
dao	**896**	1828	1817	1817	9996	5137	**4039**	4115
da2	**598**	1211	1206	1206	5701	2520	**2190**	2401
sc1	**5698**	11526	11480	11480	104369	48630	**38306**	39517
wc3-512	**1193**	2387	2387	2387	13519	7391	**6279**	6407
maze-1	**36179**	84654	84654	84654	**72356**	100580	100580	100580
maze-2	**21954**	43908	43908	43908	57261	**50930**	**50930**	**50930**
maze-4	**7906**	15812	15812	15812	24807	19453	**19095**	20930
maze-8	**2410**	4819	4819	4819	7484	5952	**5844**	6364
maze-16	**709**	1418	1418	1418	2203	1746	**1708**	1876
maze-32	**185**	370	370	370	570	459	**449**	492
room-8	**12826**	35347	35120	35120	**41211**	54528	53816	53978
room-16	**3261**	9352	9338	9338	**10735**	13884	13843	13954
room-32	**802**	2375	2375	2375	**2641**	3350	3351	3408
room-64	**192**	558	558	558	**602**	750	750	775
street256	**1881**	3840	3807	3807	27419	11281	**7788**	8689
street512	**3732**	7555	7528	7528	97202	24062	**16054**	21363

Table 4.4, we present results for the connect, search and CSR execution times. We exclude the refine time because (1) there are only subtle differences between graphs and (2) it is also included in CSR time.

- Connect phase: DSG achieves better connection times in *all*, *games* and *streets* benchmarks. SG dominates in *mazes* and *rooms*. The main differences of DSG against each type of graph are:
  - **SG**: In SG forward connect, the iteration through the main axis may have more steps than in DSG, since simple important cells are subgoals while in DSG the subgoals must have a given direction. This also translates into the amount of connections. In Figure 4.3 we can see that SG connects to every subgoal it finds, while DSG connects only to subgoals with a specific direction.



(a) DSG ignores connections in comparison to SG



(b) Paths in DSG are not always shorter than in SG

Figure 4.3. (a) In SG, s connects to every yellow subgoal, while in DSG s does not connect to any subgoal. (b) In DSG, edges from s to the green nodes represent a path of length two in SG. However, in DSG, a path from s to the red node uses the same number of edges than a path in SG.

  - **JP and JPD**: In DSG forward connect, each successful connection restricts other subsequent connections, because of the boundaries used in DIAGONAL_SCAN from Algorithm 3.2. Also each diagonal scan from a CF-scan stop independently, unlike JP and JPD where both cardinal scans continue until the diagonal scan stops. Finally, in comparison with JP target connection, the time complexity is lower, with $O(W + H)$ versus $O(WH)$.

- Search phase: JP outperforms all other algorithms in almost all categories. This is achieved since an edge in JP can represent a path of any length in SG, as explained in Section 3.1.1. It is possible to notice that DSG also outperforms SG, but the difference is lower. This occurs since a DSG edge can represent a path in SG of a length up to two, as shown in Figure 4.3.
- Connect-Search-Refine procedure: JP outperforms other algorithms in almost all categories since the search phase is predominant in comparison with the connect phase. To summarize, one can notice that JP is 2.14 times faster than DSG in the games category.

Table 4.4. Execution times of the Connect-Search-Refine procedure of the subgoal graph framework. All categories from the MovingAI benchmark are included, except *random* and *street-1024*.

	Connect (μs)				Search (μs)				CSR (μs)			
	SG	JP	JPD	DSG	SG	JP	JPD	DSG	SG	JP	JPD	DSG
all	7.14	7.31	6.21	**5.83**	1607.81	**876.94**	939.52	979.88	1647.60	**914.22**	976.17	1017.68
games-all	11.77	11.14	9.30	**8.58**	714.72	**78.57**	93.06	188.90	735.11	**95.92**	109.46	205.45
mazes-all	**1.57**	2.82	2.69	2.73	2850.88	**1892.75**	2014.37	2002.38	2916.90	**1956.69**	2078.25	2068.69
rooms-all	**1.73**	3.10	3.03	2.91	422.90	**343.22**	386.73	369.66	433.67	**354.20**	397.80	381.60
streets-all	18.04	15.25	11.61	**10.52**	466.30	**40.45**	55.38	112.74	490.25	**60.05**	72.10	128.97
bg	5.12	5.83	**3.75**	3.86	214.35	**21.57**	35.07	66.99	224.30	**29.84**	42.55	75.18
bg-512	7.84	6.15	5.24	**5.16**	191.20	**18.23**	23.61	71.72	204.81	**28.91**	33.91	82.45
dao	7.77	8.26	6.76	**6.22**	506.66	**88.52**	101.89	162.72	523.87	**103.32**	116.23	177.50
da2	5.36	5.59	**4.58**	4.59	194.94	**21.14**	28.57	56.17	206.86	**31.29**	38.48	66.73
sc1	21.33	19.46	16.44	**14.87**	1579.69	**149.56**	175.49	378.52	1613.01	**177.80**	201.79	404.58
wc3-512	7.97	9.14	8.04	**7.03**	156.65	**24.92**	28.41	49.16	169.57	**38.05**	40.70	60.72
maze-1	**1.39**	3.19	3.03	3.31	6857.02	**5546.25**	5857.16	5637.72	6973.37	**5667.13**	5976.21	5758.58
maze-2	**1.72**	3.11	3.06	3.10	4545.04	**2544.74**	2678.57	2555.02	4630.45	**2627.72**	2760.63	2638.28
maze-4	**1.60**	2.93	2.73	2.72	1710.26	**539.44**	646.57	918.13	1766.23	**587.50**	696.13	973.12
maze-8	**1.59**	2.57	2.44	2.35	532.27	**155.62**	185.62	267.21	576.11	**193.61**	224.69	309.76
maze-16	**1.61**	2.38	2.23	2.17	143.50	**41.67**	49.38	71.72	173.90	**69.66**	77.96	101.90
maze-32	**1.56**	2.31	2.16	2.15	34.43	**9.83**	11.63	17.28	52.69	**27.72**	29.67	36.00
room-8	**1.99**	4.03	3.89	3.90	1322.71	**1092.58**	1234.02	1155.49	1339.48	**1108.74**	1250.41	1173.96
room-16	**1.76**	3.24	3.12	2.96	288.75	**225.55**	251.61	254.04	299.77	**236.84**	262.93	266.28
room-32	**1.67**	2.68	2.68	2.50	76.55	**54.96**	61.79	66.71	84.91	**63.76**	70.68	76.00
room-64	**1.52**	2.47	2.44	2.32	19.72	**12.78**	14.09	16.55	26.83	**20.61**	21.92	24.55
street256	10.91	10.48	7.91	**7.38**	232.93	**31.18**	39.99	70.79	247.97	**44.54**	51.42	82.08
street512	21.59	17.63	13.46	**12.09**	582.51	**45.06**	63.04	133.63	610.89	**67.77**	82.40	152.32

4.2.2. Sugoal graph framework and Contraction Hierarchies

In this section we test the subgoal graph framework in conjunction with CH. We divided the from the preprocessing and query phases.

4.2.2.1. Preprocessing phase

In order to measure the size of a CH-subgoal graph we use the following metrics:

- **N° shortcuts**: Is the number of edges added with respect to the base graph numbers shown in Table 4.3. Each shortcut is potentially a combination of other edges.

- **% $R_{\mathcal{F}}$-reachable shortcuts**: Is the percentage of shortcuts that do not need to be unpacked, since the incident nodes holds a determined freespace reachability relation. In this case, we use is freespace-diagonal-first reachability and freespace-cardinal-first reachability. Having a higher percentage reduces the cost of the unpacking procedure.
- $|\mathbf{E}|$: Is the total number of edges in the graph after the CH preprocessing. It is the most important measure to compare graph sizes between subgoal graphs, since the number of nodes is similar.

Table 4.5 shows the preprocessing phase results of this experiment. From it, we can mention that

- **N° shortcuts:** JP is the subgoal that adds the least number of shortcuts. This occurs given that JP edges represent paths of many edges in SG or DSG. Therefore, other subgoal graphs need more shortcuts to represent JP edges, and more to represent JP shortcuts. With respect to to CH-DSG it adds the most number of shorctuts between directed subgoal graphs, due to the aforementioned reasons. This occurs more strongly in benchmarks with larger maps such as *Starcraft I* and *Street-512*.
- **% $R_{\mathcal{F}}$-reachable shortcuts**: CH-SG is the subgoal graph with highest percentage of $R_{\mathcal{F}}$-reachable shortcuts, followed by CH-DSG and CH-JPD in *all* benchmarks. The percentage of $R_{\mathcal{F}}$-reachable shortcuts $X\%$ can be interpreted as that $X\%$ of the shortcuts can be treated as edges in the original graph and only $1 - X\%$ of the shortcuts actually need the CH unpacking procedure. CH-DSG has almost 70% of $R_{\mathcal{F}}$-reachable shortcuts, therefore this optimization is essential for speeding up the refine procedure. The percentage is still important in CH-JPD, reaching 48.1%. In CH-JP is where this optimization is least important.

Table 4.5. CH subgoal graph sizes in the MovingAI benchmark. All categories are included except *random* and *street-1024*.

	N° shortcuts				% $R_{\mathcal{F}}$-reachable shortcuts				$\mid E \mid$			
	SG	JP	JPD	DSG	SG	JP	JPD	DSG	SG	JP	JPD	DSG
all	**2739**	3293	3646	4460	**75.63**	22.13	48.10	69.82	15309	16175	**14402**	15821
games-all	1560	**1086**	1375	1923	**84.30**	25.12	54.54	76.72	12320	11102	**9299**	10228
mazes-all	**6638**	13536	13678	14064	**29.33**	5.23	15.97	26.56	**20362**	43389	43445	44260
rooms-all	**9707**	17384	17880	21779	25.80	1.67	4.56	**34.36**	**16606**	35512	35820	39808
streets-all	4592	**3117**	4155	5673	**78.69**	26.34	52.49	75.91	35747	20788	**16076**	20699
bg	448	**231**	388	570	**84.84**	18.59	66.17	79.38	2602	2083	**1694**	2018
bg-512	676	**375**	504	871	**83.65**	24.51	53.70	77.90	4758	2949	**2628**	3832
dao	975	**747**	927	1209	**83.50**	25.06	50.88	73.89	5973	5884	**4966**	5323
da2	481	**279**	388	568	**85.70**	28.65	54.88	77.40	3332	2799	**2578**	2969
sc1	6526	**4582**	5691	8029	**85.79**	35.32	52.18	79.49	58711	53211	**43998**	47547
wc3-512	1298	**1099**	1262	1527	**81.65**	20.62	37.61	70.58	8058	8490	**7541**	7933
maze-1	**21808**	52107	52107	52099	4.75	7.47	7.47	**7.48**	**57986**	152687	152687	152680
maze-2	**12359**	22134	22134	22134	**15.39**	5.05	5.05	5.04	**40990**	73064	73064	73063
maze-4	**4040**	4976	5597	7234	**38.42**	4.71	19.69	34.53	**16443**	24429	24692	28163
maze-8	**1200**	1487	1653	2151	**38.17**	5.08	19.25	34.99	**4943**	7439	7498	8516
maze-16	**346**	422	471	628	**37.91**	4.62	20.51	36.10	**1448**	2168	2180	2504
maze-32	**75**	87	103	141	**41.33**	4.47	23.86	41.22	**360**	546	552	633
room-8	**30206**	55628	57391	68383	24.03	2.98	8.09	**26.02**	**50811**	110156	111207	122361
room-16	**7016**	11585	11772	15258	23.50	1.73	4.66	**30.07**	**12384**	25469	25615	29212
room-32	**1400**	2048	2075	3014	24.86	0.75	2.53	**36.30**	**2721**	5398	5426	6422
room-64	**206**	276	280	462	30.80	1.23	2.96	**45.05**	**507**	1026	1030	1237
street256	2809	**2265**	2944	3818	**76.87**	22.63	49.86	72.81	16518	13546	**10732**	12507
street512	6375	**3968**	5365	7529	**80.51**	30.06	55.11	79.01	54976	28030	**21419**	28892

- **$|\mathbf{E}|$:** Despite JP being the subgoal graph with the least number of added shortcuts, each subgoal graph stays in the same place overall, but the differences narrow. CH-DSG now has 10% more edges than CH-JPD in *games* benchmark.

4.2.2.2. Query phase

In order to present the differences in the query speed, we present the following metrics:

- **Search time:** The search stage was the most time consuming phase in Section 4.2.1.2 and reducing it is one of the main objectives of CH.
- **Refine time:** The refine time in CH-subgoal graphs now considers both the time of the unpacking stage of CH and the time of the refine stage of subgoal graphs.
- **CSR time:** Similar to CSR time in Section 4.2.1.2, it is used to determine the fastest algorithm in the query phase.

- **N° exp.:** The number of expansions performed by Bidirectional A*. It includes forward and backward expansions.
- **Successors per exp.:** The average number of nodes added (or updated) to the Open queue after an expansion.

Since the connect procedure remains almost the same than in the base subgoal graph, we do not include *connect time*. However, there is a subtle and even increase in connection times with respect to the base subgoal graph, given that each edge in CH is represented by more attributes. In any case, the influence of the connect time can be seen reflected in CSR time.

Table 4.6 presents the results of solving all problems in MovingAI, whereas Table 4.7 presents the search stats. Here, we can observe that:

(i) **Search time:** Unlike Section 4.2.1.2, now JPD achieves the best CSR time performance overall. However, the panorama is much more competitive, with CH-SG winning in 2 subcategories, CH-JP in 3, CH-JPD in 9 and CH-DSG in 4. Here, we observe that differences against CH-DSG are almost nonexistent, averaging 2.3%.

(ii) **Refine time:** CH-JP achieves the smallest refine times. This can be explained since CH-JP add the least number of shortcuts as discussed in Section 4.2.2.1. However, as other subgoal graphs use $R_{\mathcal{F}}$-reachable shortcuts, the differences in the refine time are small. For example CH-DSG has 35% more shortcuts but only 3.2% slower refine.

(iii) **CSR time:** CH-JPD is the fastest subgoal graph in MovingAI, however, the differences are small: CH-JP and CH-DSG are 3.1% and 1.6% slower respectively. With respect to CH-DSG, it leads the CSR time in 5 subcategories, in comparison with the 4 subcategories in the search phase. This can be explained since DSG is the fastest algorithm in the connect phase Section 4.2.1.2.

(iv) **N° Exp. and successors per exp.:** We can notice that CH-JP is the algorithm with the fewest expansions, but DSG is the algorithm with lower successors per

Table 4.6. Execution times of the Connect-Search-Refine procedure of the CH subgoal graph framework. All categories from the MovingAI benchmark are included, except *random* and *street-1024*.

	Search (μs)				Refine (μs)				CSR (μs)			
	SG	JP	JPD	DSG	SG	JP	JPD	DSG	SG	JP	JPD	DSG
all	60.22	21.13	**20.46**	20.94	70.01	**55.09**	56.36	56.86	139.20	84.56	**84.04**	84.73
games-all	98.10	26.56	**25.22**	25.80	9.82	**8.67**	8.90	9.22	122.75	47.93	**44.94**	45.24
mazes-all	12.01	9.85	**9.82**	10.02	148.34	**115.60**	118.24	119.01	162.26	**128.66**	131.15	132.15
rooms-all	59.55	60.64	60.22	**58.41**	16.97	**13.85**	14.07	14.16	79.11	78.57	78.32	**76.59**
streets-all	126.04	23.40	**22.97**	27.02	6.81	**5.96**	6.11	6.33	155.28	46.36	**42.25**	46.13
bg	41.61	12.12	**11.31**	13.25	4.93	**3.86**	4.15	4.33	53.16	22.59	**19.99**	22.41
bg-512	45.51	10.64	**10.50**	13.71	7.30	**6.62**	6.83	7.11	62.47	24.11	**23.17**	26.89
dao	70.21	25.36	23.87	**23.69**	10.91	**9.45**	9.65	9.94	90.95	44.18	41.32	**41.12**
da2	42.97	11.53	**11.29**	13.02	8.01	**6.77**	7.04	7.40	57.85	24.59	**23.70**	25.89
sc1	191.25	46.79	44.12	**43.11**	12.86	**11.52**	11.79	12.22	230.93	80.66	75.20	**73.01**
wc3-512	45.85	16.81	**16.58**	17.14	6.59	**5.96**	6.11	6.25	62.69	33.20	31.95	**31.76**
maze-1	**11.19**	13.51	13.31	13.57	306.75	**234.19**	238.57	237.93	319.60	**251.52**	255.60	255.46
maze-2	15.12	11.98	**11.80**	11.83	204.70	**156.44**	159.46	157.92	221.77	**172.05**	174.79	173.37
maze-4	13.72	**10.02**	10.20	10.28	101.75	**78.41**	81.28	85.24	117.39	**91.60**	94.47	98.48
maze-8	12.03	8.24	**8.22**	8.49	72.19	**59.68**	61.55	63.74	86.18	**70.69**	72.45	74.81
maze-16	10.20	**6.39**	6.52	6.77	47.09	**42.11**	43.92	44.76	59.34	**51.13**	52.94	53.98
maze-32	7.83	**4.82**	4.87	5.23	25.77	**25.07**	25.08	25.76	35.66	32.56	**32.49**	33.43
room-8	**137.19**	150.86	149.56	142.84	29.18	**22.34**	22.90	23.06	**169.71**	179.76	178.86	172.44
room-16	62.21	59.42	59.26	**58.56**	17.79	**14.60**	14.63	14.61	82.63	77.96	77.77	**77.07**
room-32	28.98	25.54	25.46	**25.11**	12.05	**10.37**	10.59	10.58	43.32	39.05	39.20	**38.70**
room-64	12.23	9.25	**9.10**	9.56	9.26	**8.37**	8.46	8.69	23.64	20.40	**20.31**	20.92
street256	68.88	20.33	**19.20**	21.62	5.14	**4.22**	4.37	4.60	87.95	36.47	**33.01**	35.36
street512	154.51	24.93	**24.85**	29.71	7.64	**6.82**	6.98	7.19	188.81	51.28	**46.86**	51.50

exp, i.e., a lower branching factor. These two effects combined even out the search times, as shown in Table 4.6.

In games benchmarks, CH-DSG is the fastest algorithm in **dao**, **sc1** and **wc3-512**. The differences with CH-JPD are small, but in Starcraft I CH-DSG is 3.0% faster than CH-JPD. These benchmarks are characterized by its large dimensions and large problems, as discussed in Section 4.1.1. Therefore, we present a detailed analysis in order to determine if the size of the map is correlated with CH-DSG better performance. For that purpose, we compare the overall dominant algorithm CH-JPD with CH-DSG in these benchmarks. In this comparison, we contrast the CSR times of each algorithm with the average graph size. In order to express this in percentage and to have positive values when CH-DSG is faster we use the following *percentage improvement* formula:

$$\text{Percentage improvement} = 100 \times (\frac{\text{CH-JPD CSR time}}{\text{CH-DSG CSR time}} - 1)\%$$

Table 4.7. Statistics of the bidirectional search performed by CH subgoal graphs. All categories from the MovingAI benchmark are included, except *random* and *street-1024*.

	N° exp.				Successors per exp.			
	SG	JP	JPD	DSG	SG	JP	JPD	DSG
all	45.92	**33.44**	34.19	36.58	2.09	1.51	1.47	**1.38**
games-all	60.81	**35.52**	36.46	40.12	2.63	1.74	1.67	**1.54**
mazes-all	29.27	**29.04**	29.55	30.20	1.06	**0.97**	**0.97**	**0.97**
rooms-all	**47.21**	60.44	60.65	61.62	2.32	2.06	2.05	**1.97**
streets-all	53.05	**23.07**	24.57	31.66	5.37	3.23	3.04	**2.49**
bg	34.55	**20.36**	21.61	24.85	2.28	1.59	1.44	**1.40**
bg-512	34.87	**18.42**	19.38	23.72	2.52	1.63	**1.55**	**1.55**
dao	52.74	**37.14**	37.73	39.96	2.34	1.59	1.51	**1.42**
da2	39.36	**22.67**	23.35	26.98	2.02	1.35	1.32	**1.27**
sc1	100.72	**54.81**	56.08	60.71	3.06	1.91	1.84	**1.63**
wc3-512	33.99	**21.99**	22.67	25.47	3.05	2.40	2.31	**1.98**
maze-1	**29.56**	37.15	37.15	37.74	1.04	**0.98**	**0.98**	**0.98**
maze-2	35.74	**34.97**	35.03	35.40	1.07	**0.98**	**0.98**	**0.98**
maze-4	32.04	**29.71**	30.72	31.19	1.08	**0.97**	**0.97**	0.98
maze-8	28.61	**25.48**	26.27	27.22	1.08	0.97	**0.96**	0.97
maze-16	24.71	**20.72**	21.80	22.29	1.07	**0.96**	**0.96**	**0.96**
maze-32	19.57	**15.99**	16.45	17.76	1.05	**0.95**	**0.95**	0.96
room-8	**84.02**	121.55	122.22	120.93	3.24	2.77	2.74	**2.66**
room-16	**53.16**	65.49	65.63	67.91	2.57	2.27	2.27	**2.16**
room-32	**33.65**	37.85	37.98	39.32	1.99	1.80	1.80	**1.73**
room-64	19.54	19.09	**19.00**	20.53	1.55	1.45	1.44	**1.39**
street-256	38.95	**21.76**	22.53	27.60	4.16	2.99	2.76	**2.33**
street-512	60.07	**23.73**	25.59	33.68	5.98	3.35	3.17	**2.57**

In order to normalize graph sizes, we use the average number of edges between CH-JPD and CH-DSG.

$$\text{Average graph size} = \frac{|E|_{\text{CH\_JPD}} + |E|_{\text{CH\_DSG}}}{2}$$

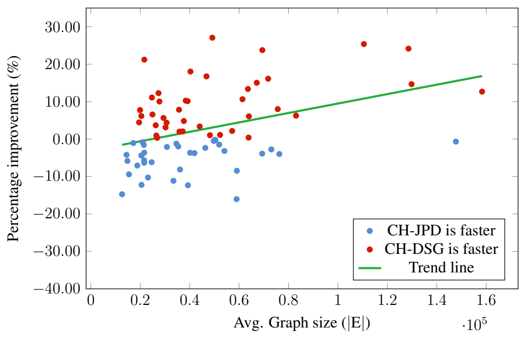
We also include **bg-512** in this comparison since it also contains large maps.

In Figure 4.4 we plot the percentage improvement of CH-DSG versus the average graph size, and also add a trend line. Here, we can see that all trend lines have a positive slope, i.e., the speedup that DSG grants with respect to JPD increases with the graph size. However, the slope is little steep and there are many outliers. In (a), (b) and (c) we can see that CH-JPD dominates in smaller graphs, but as the graph size increases CH-DSG takes the lead. In (d), the speedup favors CH-JPD over all graph sizes, however, the slope is still positive, reducing the difference with CH-DSG as the graph size increases. The maps with highest percentage improvement for CH-DSG in these benchmarks reach 27%, 20%, 22% and 8% in sc1, dao, wc3-512 and bg-512 benchmarks respectively, representing a

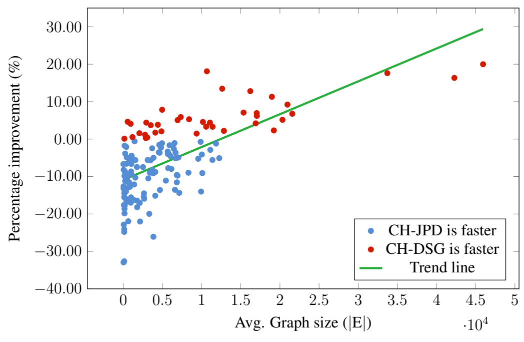
huge improvement for the state-of-art algorithms. This becomes more relevant when we consider that these benchmarks are games benchmark with larger graphs and problems with higher costs.
To understand why this happen, we show in Figure 4.5 the maps on which CH-DSG performs best for each respective benchmark. Based on this, we made the following observations:

- **There is a large number of connected components of obstacles:** In (a) Expedition, we can find a lots of medium-size isles and also, in the center of the map, a large number of small blocks. This situation repeats in (b) ost100d, where there are a great number of isolated group of pixels. In (c) stromguarde, we can see a large number of blocks and in (d) AR0414SR, there are several medium-size islands.
- **The traversable space allows long diagonal movements:** This is crucial to understand why CH-DSG outperforms CH-JPD on these maps. Long diagonal movements among with large number of connected components results in that, during a forward connect in JPD, the diagonal scan takes longer to finish, at the same time it detects a large number of connected components with its respective jump points, resulting in more connections, for both the *Connect graph* in the preprocessing stage and the *Connect* in the query phase.

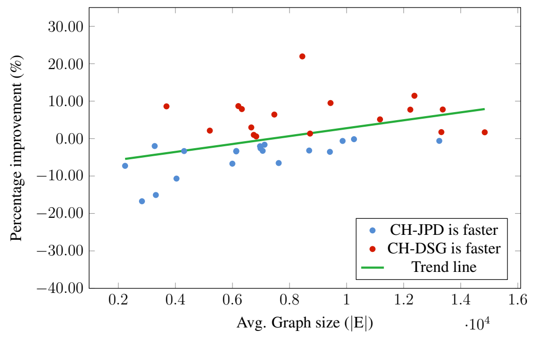
Despite the aforementioned, DSG has 1.8% to 3% more edges than JPD in these benchmarks, as shown in Table 4.3. However, this difference is small and many of these edges are incident to subgoals in rough walls. Since subgoals in rough walls are mainly used to navigate through the wall, they often have low hierarchy levels and therefore most of these edges are skipped in a bidirectional search over the CH graph. In conclusion, CH takes away most of DSG weakness on these type of maps whereas JPD maintains higher connection times and similar search times. We can observe that the larger size of the graph has a positive correlation with CH-DSG better performance, but we think that the aforementioned observations plays a more important role.



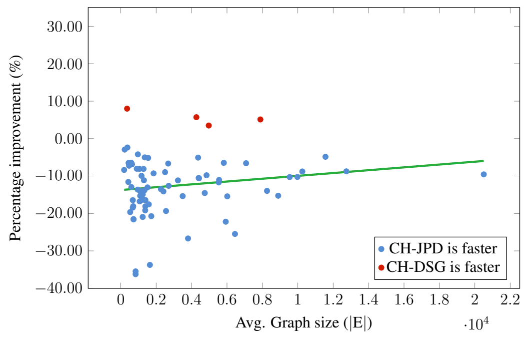
(a) Starcraft



(b) Dragon Age: Origins



(c) Warcraft III - 512



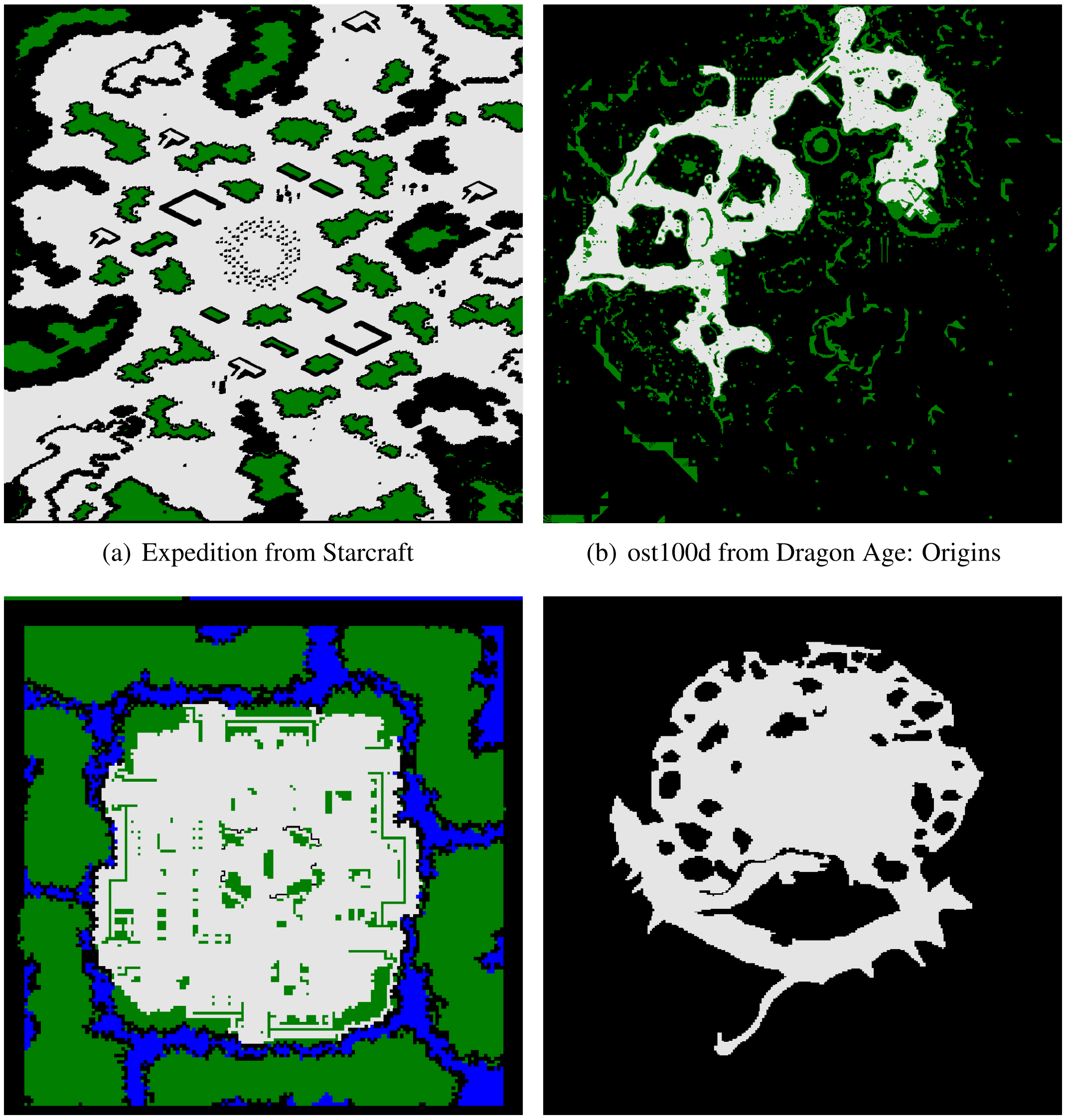
(d) Baldurs gate II - 512

Figure 4.4. Scatter plot of the percentage improvement of CH-DSG over CH-JPD versus average graph size in games benchmark

4.2.3. Memory analysis

In Section 4.2.2.2 we showed that the differences in execution time between directed subgoal graphs are small, with CH-DSG being 1.6% and 0.2% slower than CH-JPD in *all* and *games-all* benchmarks respectively. Therefore, other factors, such as memory usage, gain more relevance in order to determine which graph is more appropriate for each context. For the following comparisons, we exclude SG and CH-SG, since the benchmarks in which DSG is faster or is close to being, its main competitors are JP and JPD.

That said, in Section 3.1.3.4 we explain how DSG can use half of the clearances that JP and JPD use. In order to quantify the improvements this represents, we summarize all the information that must be saved after the preprocessing stage:



(a) Expedition from Starcraft (b) ost100d from Dragon Age: Origins

(c) stromguarde from Warcraft III-512 (d) AR0414SR from Baldurs Gate II-512

Figure 4.5. Maps in which CH-DSG performs the best

- **Clearances**: We use 1 byte for each clearance value. Therefore, considering all directions we have $8 \times W \times H$ bytes per set of diagonal and cardinal clearances. Thus SG and DSG use a total of $8 \times W \times H$ bytes for clearance memory and JP and JPD use a total of $16 \times W \times H$ bytes.
- **Subgoal graphs**:

    - Information per subgoal: It uses 4 bytes for the position and direction, 1 byte for subgoals that reference it and 6 bytes for edges information. Total: 11 bytes.
    - Information per edge: It uses 4 bytes for the target nodes. Total: 4 bytes.

  This adds up to a total of $11 \times |V| + 4 \times |E|$ bytes.
- **CH subgoal graphs:**
    - Information per subgoal: Additional to the 11 bytes for the subgoal information, we use an additional 6 bytes for reverse edges and 1 byte for avoidance information. Total: 18 bytes.
    - Information per edge: We can avoid saving the cost and unpacking information for edges that do not need to be unpacked. These edges are edges which are not shortcuts or shortcuts that are freespace-R-reachable. To do so, for every edge we save the source and target node and a new *unpacking identifier*, which is undefined (-1) for those edges (total of 12 bytes). For the remaining $|E'|$ edges, we save the unpacking information in another vector: 4 bytes for the cost and 8 bytes for the base edges. This vector is referenced using the unpacking identifier. Total: $12 \times |E| + 12 \times |E'|$ bytes.

  This adds up to a total of $18 \times |V| + 12 \times |E| + 12 \times |E'|$ bytes .

In Table 4.8 we show the memory usage of each directed subgoal graph and the percentage of that memory that is used on clearances, signaled as $\%$ Clr Mem. This way we can measure the benefit of using DSG in terms of memory. Dark orange rows represent benchmarks with the highest memory usage, whereas dark blue rows represent benchmarks where the graph weights more with respect to the weight of the clearances.

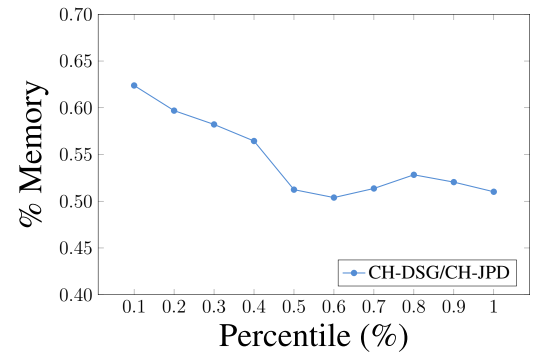
- **Subgoal graphs:** The benchmark with the highest memory usage is *sc1*, but it presents a high percentage of clearances. Therefore, DSG substantially reduces memory usage to 51.9% with respect to JPD. In general terms, DSG uses 51.7% and 51.0% of JP and JPD memory on *all* and *games-all* benchmarks. This occurs since the percentage of memory that corresponds to clearances is greater than

Table 4.8. Memory usage of directed subgoal graphs and its CH versions.

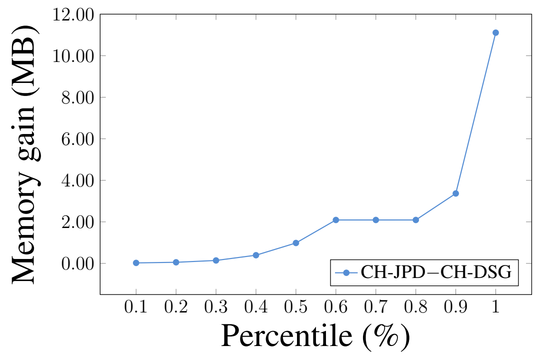
	Mem. (MB)			% Clr Mem.			Mem. CH (MB)			% Clr Mem. CH		
	JP	JPD	DSG	JP	JPD	DSG	JP	JPD	DSG	JP	JPD	DSG
all	3.18	3.17	1.64	96.39	96.66	93.39	3.40	3.38	1.86	90.02	90.64	82.27
games-all	2.97	2.96	1.51	97.54	97.82	95.63	3.10	3.08	1.64	93.49	94.19	88.57
mazes-all	4.60	4.60	2.50	91.31	91.32	83.96	5.35	5.35	3.27	78.39	78.39	64.26
rooms-all	4.40	4.40	2.30	95.34	95.37	91.14	5.06	5.06	3.00	82.99	83.02	70.04
streets-all	2.76	2.74	1.44	95.11	95.93	91.38	3.01	2.95	1.69	87.08	88.88	77.72
bg	0.29	0.29	0.15	93.46	94.24	88.78	0.32	0.31	0.18	84.56	86.09	74.05
bg-512	4.22	4.22	2.13	99.38	99.42	98.69	4.27	4.26	2.18	98.39	98.49	96.41
dao	2.19	2.19	1.11	98.08	98.29	96.61	2.27	2.26	1.18	94.86	95.38	90.98
da2	2.45	2.45	1.24	98.99	99.05	98.05	2.49	2.49	1.28	97.45	97.57	94.94
sc1	7.47	7.43	3.86	95.68	96.22	92.60	8.05	7.93	4.39	88.85	90.13	81.48
wc3-512	4.25	4.25	2.15	98.66	98.76	97.53	4.36	4.35	2.25	96.28	96.55	93.31
maze-1	5.54	5.54	3.45	75.70	75.70	60.91	8.23	8.23	6.13	50.98	50.98	34.21
maze-2	4.89	4.89	2.79	85.92	85.92	75.31	6.16	6.16	4.07	68.08	68.08	51.61
maze-4	4.45	4.45	2.36	94.31	94.34	89.02	4.85	4.85	2.80	86.50	86.50	75.00
maze-8	4.28	4.27	2.18	98.17	98.18	96.34	4.40	4.40	2.31	95.40	95.41	90.71
maze-16	4.22	4.22	2.12	99.43	99.44	98.85	4.26	4.26	2.17	98.54	98.54	96.94
maze-32	4.20	4.20	2.11	99.83	99.83	99.65	4.22	4.22	2.12	99.55	99.55	99.05
room-8	4.81	4.80	2.70	87.31	87.42	77.64	6.84	6.84	4.85	61.32	61.39	43.30
room-16	4.36	4.36	2.26	96.33	96.34	92.92	4.82	4.82	2.76	87.05	87.06	76.06
room-32	4.24	4.24	2.14	99.04	99.04	98.08	4.34	4.34	2.25	96.80	96.79	93.34
room-64	4.21	4.21	2.11	99.75	99.75	99.50	4.23	4.23	2.13	99.27	99.27	98.45
street-256	1.14	1.13	0.60	92.21	93.40	87.09	1.31	1.27	0.77	80.10	82.48	68.72
street-512	4.38	4.35	2.27	95.87	96.59	92.52	4.72	4.63	2.61	89.02	90.64	80.35

90%, thus, the memory usage is dominated by clearances. The exceptions are dark blue rows as *maze-1* or *room-8*, where the weight of the graph is comparable to the weight of the clearances.

- **CH subgoal graphs:** The memory benefits are greater on games maps (light blue), where CH-DSG uses 53.2% of the memory used by CH-JPD. In *all* benchmark, the benefit reaches 55.0%. The memory benefits are still important in *streets*, using only a 57.3% of CH-JPD's memory. *maze-1* and *room-8* are the benchmarks that use the most memory. Given that they present a lower percentage of clearance memory, the benefit of using DSG is also lower. This occurs since CH adds many shortcuts and thus increases the relative weight of the graph with respect to the clearances.



(a) Percentage of memory used by CH-DSG with respect to CH-JPD over percentile of total memory usage.



(b) Net memory gain of CH-DSG vs CH-JPD over percentile of total memory usage.

Figure 4.6. Memory comparison between CH-DSG and CH-JPD.

4.2.3.1. Memory usage and graph size in *games*

Since the memory benefits of using DSG or CH-DSG are greater in *games* benchmark we compare it with the dominant algorithm, CH-JPD. For this purpose, we include the total size as a new dimension. For each graph we sort the maps by their total memory usage (clearances and graph) in ascending order. Then, for each percentile of these sorted maps we compare the memory usage of CH-DSG with CH-JPD. Figure 4.6 (a) shows the percentage of memory that CH-DSG uses with respect to CH-JP and (b) shows the net memory gain. Here, we can see that the memory benefits of using CH-DSG are greater when the total memory used is higher. In maps with lower memory usage the percentage of memory used by CH-DSG reaches 62% which translates into gains of 0.02 megabytes per map and in maps with higher memory usage the percentage of memory used by CH-DSG reaches 51% which translates into gains of up to 11 megabytes per map.

4.2.4. Avoidance

The avoidance utilized for subgoal graphs consist of backward connection avoidance and forward search avoidance, whereas the avoidance used for CH subgoal graphs consists only of backward connection avoidance, since avoidable nodes tend to have lower

Table 4.9. Search statistics when using avoidance. All instances from the 10 maps in lexicographical order from games and streets categories of MovingAI benchmark are included (excluding street-1024).

	Without avoidance			With avoidance		
Graph	Added edges	N° exp.	CSR time (μs)	Added edges	N° exp.	CSR time (μs)
DSG	31.32	406.29	217.85	27.85	338.02	190.15
JP	59.57	196.02	113.36	24.18	138.14	83.81
JPD	40.96	232.13	122.09	26.95	177.15	94.85
CH-DSG	31.32	40.18	46.58	27.85	38.14	44.56
CH-JP	59.57	47.14	61.02	24.18	31.96	45.84
CH-JPD	40.96	40.23	49.78	26.95	33.29	43.37

hierarchies and thus are less likely to being added to the open. In order to measure the impact of avoidance, we perform the following experiment: For a subset of the first 10 maps (lexicographically) from each *game* and *street* subcategories, we present statistics with and without avoidance. These statistics are:

- Added edges: The average number of edges added in the connection procedure.
- N° expansions: The average number of expansions during the search.
- CSR time: The average total execution time for each instance.

The results of this experiment are shown in Table 4.9.

We can notice that the amount of edges added in the connection procedure is drastically reduced with the connection procedure, reaching a 59% reduction in JP. However we can also notice that this improvement is lower in DSG, where the reduction is only 11%. It is also important to notice that without avoidance, CH-DSG is indisputably the fastest search algorithm.
We discovered that in JP about 50% of subgoals are backward avoidable, while in JPD and DSG this percentage is about 30% and 12% respectively. Forward avoidance is more even between the different graphs, but also shows a similar tendency. In summary, avoidance represent an improvement to all directed subgoal graphs, but is more crucial in JP and JPD.

5. CONCLUSIONS

5.1. Conclusions

DSGs are an extension to the subgoal graph framework, in which we expanded the usage of the incoming direction first introduced in JP. DSG is a more conservative subgoal graph in the sense that it introduces edges that are relatively short in comparison with JP or JPD. In addition, its connection procedure is the fastest across the MovingAI benchmark while also using half of the memory. This connection procedure also explores the grid in a novel cardinal-first approach. The above benefits come at the cost of slower search times, mainly due to the fact that solutions have a higher depth. However, this pitfall can be largely overcome by using CH-DSG, which generates a graph that is as fast as its competitors, improving the state-of-art query times in several benchmarks, mainly in games and rooms. The speedups in these benchmarks reach up to 3.0%.

We also characterized the scenarios where CH-DSG outperforms its competitors the most, these are mainly in large maps in which the traversable space allows long diagonal movements where there also are a large number of connected components. In these scenarios, speedups with respect to the state-of-art CH-dsgs can reach up to 27%. In addition to the above, CH-DSG still uses considerably less memory than other CH-dsgs, using in average 45% less memory in all benchmarks and 47% less memory in games benchmarks.

In the process of developing DSG, we provided an in depth analysis of the subgoal graph framework, possibly improving the explainability of each one of its components, and providing several optimization for directed subgoal graphs and CH-dsgs. These optimizations are (1) the extension of the notion of avoidance, in which a group of nodes can be ignored in both search and connection stages (2) the improvement of suboptimal approaches to reduce redundant edges in CH-dsgs and (3) the generalisation of the usage of freespace-reachable shortcuts in CH-sgs to reduce refine times. DSG represent an extension to the subgoal graph framework that in overall widens the spectrum of scenarios in which subgoal graphs are the most situable preprocessing technique for solving path planning problems.

5.2. Future work

Through the development of this research, we identified several other opportunities to improve the subgoal graph framework.

- We provided a new subgoal graph that is based on SG safe-freespace reachability, however, we added a diagonal-first condition to enhance solution depths and branching factor. Given this, it is possible to build a DSG without this condition that would be more conservative than DSG and therefore it could present other benefits yet undiscovered.
- Considering there are several different subgoal graphs, one could think building a meta-subgoal graph, which determines in preprocessing time what would be the best subgoal graph for that grid graph, this way obtaining most of the benefits for each scenario.
- With respect to the subgoal graph framework, it is possible to expand the avoidance concept to nodes that only reach (or are reached) by avoidable nodes, therefore generating avoidance trees. This could allow to increase the amount of avoidable nodes at the cost of an overhead in the connection phase depending on the height of the trees. It should be possible to limit height of trees and therefore get performance benefits.
- In CH subgoal graphs, it would beneficial to include a priority term that considers the probability of a subgoal being connected to the start and goal. This could be done by counting the number of cells that are direct-R reachable to the subgoal or that direct-R reach the subgoal.
- We think that an improvement to the rooms benchmark would be to add doors of different sizes, since doors of size 1 favors SG to the detriment of any subgoal graph with more than one node at each cell.

REFERENCES

Abraham, I., Delling, D., Fiat, A., Goldberg, A. V., & Werneck, R. F. (2016, dec). Highway dimension and provably efficient shortest path algorithms. *J. ACM*, *63*(5). Retrieved from `https://doi.org/10.1145/2985473` doi: 10.1145/2985473

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on experimental algorithms* (p. 319–333). Berlin, Heidelberg: Springer-Verlag.

Harabor, D., & Grastien, A. (2011). Online graph pruning for pathfinding on grid maps. In *Proceedings of the twenty-fifth aaai conference on artificial intelligence.* Retrieved from `http://www.aaai.org/Conferences/AAAI/aaai11.php`

Harabor, D., Uras, T., Stuckey, P., & Koenig, S. (2019). Regarding jump point search and subgoal graphs. In *Proceedings of the international joint conference on artificial intelligence (ijcai) 2019.* Retrieved from `https://ijcai19.org/,https://www.ijcai.org/proceedings/2019/`

Rivera, N., Hernández, C., Hormazábal, N., & Baier, J. (2020, January 1). The 2k neighborhoods for grid path planning. *Journal of Artificial Intelligence Research*, *67*, 81–113. doi: 10.1613/jair.1.11383

Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(2), 144-148. doi: 10.1109/TCIAIG.2012.2197681

Uras, T. (2019). *Speeding up path planning on state lattices and grid graphs by exploiting freespace structure* (Ph.D. in Computer Science, University of Southern California).

Retrieved from `http://idm-lab.org/`

Uras, T., & Koenig, S. (2018). Understanding subgoal graphs by augmenting contraction hierarchies. In *Proceedings of the 27th international joint conference on artificial intelligence* (p. 1506–1513). AAAI Press.

Uras, T., Koenig, S., & Hernández, C. (2013). Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the twenty-third international conference on international conference on automated planning and scheduling* (p. 224–232). AAAI Press.