The Grid-Based Path Planning Competition: 2014 Entries and Results

Nathan R. Sturtevant
University of Denver
Denver, CO, USA
sturtevant@cs.du.edu

Jason Traish
Charles Sturt University
Bathurst, Australia
jtraish@csu.edu.au

James Tulip
Charles Sturt University
Bathurst, Australia
jtulip@csu.edu.au

Tansel Uras
University of Southern California
Los Angeles, USA
turas@usc.edu

Sven Koenig
University of Southern California
Los Angeles, USA
skoenig@usc.edu

Ben Strasser
Karlsruhe Institute of Technology
Karlsruhe, Germany
strasser@kit.edu

Adi Botea
IBM Research
Dublin, Ireland
adibotea@ie.ibm.com

Daniel Harabor
NICTA
Eveleigh, Australia
daniel.harabor@nicta.com.au

Steve Rabin
DigiPen Institute of Technology
Redmond, WA, USA
steve.rabin@gmail.com

Abstract

The Grid-Based Path Planning Competition has just completed its third iteration. The entries used in the competition have improved significantly during this time, changing the view of the state of the art of grid-based pathfinding. Furthermore, the entries from the competition have been made publicly available, improving the ability of researchers to compare their work. This paper summarizes the entries to the 2014 competition, presents the 2014 competition results, and talks about what has been learned and where there is room for improvement.

Introduction and Background

Grid-based path planning is a broadly studied topic, with a large variety of published approaches across many application domains. For many years work in this domain lacked broadly distributed standard benchmark problems for comparing work and standard implementations for comparing the quality of work. The first author of this paper worked to fix both of these situations by (1) introducing a variety of standard benchmark problems that can be used to test programs that run on grid-based maps and (2) by starting the grid-based path planning competition (GPPC).

Citation counts suggest that the pathfinding benchmarks have been broadly adopted. (See (Sturtevant 2012) for the origin of each of the maps in the benchmark set.) The GPPC competition has run three times, in 2012, 2013, and 2014. Over this time period there has been a significant increase in the performance of entries to the competition, and the understanding of the structure of grid-based maps has also improved.

This paper describes in detail the methodology used for the competition and the results from the 2014 competition. Following this, the authors of each of the 2014 entries describe their entries. The paper concludes with a discussion of future directions for the competition and for grid-based pathfinding research.

Competition Setup

The competition is set up to experiment on a broad range of problem instances and a broad range of map sizes.

The GPPC runs on a set of 132 maps taken from several map sources. The source of each map set, and the number of maps in each set are shown in Table 1. Sample maps from the competition are shown in Figure 1.

Table 1: Map distribution in the GPPC.

Source	# Maps	# Problems
Starcraft	11	29,970
Dragon Age 2	57	54,360
Dragon Age: Origins	27	44,414
Mazes	18	145,976
Random	18	32,228
Rooms	18	27,130
Total	132	347,868

Prior to the 2012 competition the maps from Dragon Age 2 were not publicly available. These have been used in the GPPC competition but have not been released as part of the general online pathfinding map repository, although there are plans to do so in the future. The maps from Starcraft and Dragon Age: Origins are part of the pathfinding repository and use the standard problems created for these maps, although these are only a subset of the total available maps. The random, room, and maze maps are artificial. These vary in size from 100×100 to 1550×1550. The random maps have exactly 33% of the cells blocked, and then everything except the largest connected component in the map is removed, removing approximately another 5% of the cells in

each map. The room maps contain rooms that are 10% of the size of the map. Originally all rooms are closed, but doors are opened between rooms randomly. When a wall is opened, $1/8$ of the cells in the wall are randomly unblocked. Figure 1 shows (a) 10×10 rooms in a 100×100 map and (b) 40×40 rooms in a 400×400 map. The maze maps are mazes with a corridor size which is 1% of the size of one dimension of the map. So, the 100×100 map has corridors of size 1, and 800×800 map has corridors size 8. Figure 1 shows (c) a portion of a 100×100 maze and (d) a portion of a 400×400 maze.

Problems for testing are selected randomly. Problems are divided into buckets according to the length of the optimal solution divided by 4. So, all problems with optimal length $[4, 8)$ are in the same bucket. 10 problems are selected for each bucket, and the number of buckets per map is determined by the length of the paths in a particular map. There are more problems on the maze maps because the paths in mazes are so long. Some Dragon Age 2 maps are relatively small, so even though there are a large number of maps, the number of problems per map is smaller than other maps.

The distribution of map sizes per map source is shown in Figure 2. The map sizes range over 4 orders of magnitude from the largest to the smallest. The smaller maps are from the Dragon Age games, while the larger maps are from Starcraft and the artificial maps.

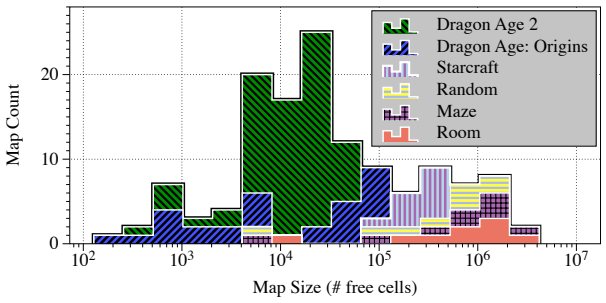


Figure 2: A histogram of the map sizes used in the competition.

Experimental Setup

To ensure that the competition results are not overly influence by CPU timing variations, we run the problem set 5 times on each entry. Entries must implement a simple interface for pre-computing data for a map, loading the pre-computed data, and doing a path computation on a map. Pre-computation is performed for all entries before they are run. The repeated runs are not consecutive, so the data from one run will not be in cache on a subsequent run. Entries can choose to return the entire path in one step, or can do so incrementally, creating a real-time agent that can interleave planning and movement.

In all years of the competition the experiments have been run on the same machine, a server with a 2-processor quad-core Xeon E5620 running at 2.4Ghz with 12 GB of RAM. The solving process uses a sequential API, but pre-processing can be done in parallel.

Depending on the domain, there are a large number of interesting metrics that determine the best entry. We do not set a particular utility function to determine the best entry, but instead look for the Pareto optimal frontier of non-dominated entries. This allows a practitioner to analyze the data for themselves, choosing the method that is best suited for their own application.

The metrics measured include:

- Total Time (seconds): This is the total time to find the solution to all problems.
- Average Time (ms): This is the average time in milliseconds to find a single path.
- 20 Steps (ms): This is the average time in milliseconds to find the first 20 steps of a path. This measures how quickly a path is available to follow, which is important in real-time applications such as games or robotics.
- Max. segment (ms): This is the average of the maximum time required to produce any individual segment of a path. This measures the worst-case real-time performance.
- Average Length: This is the average length of a returned path. If an entry is optimal on long paths and suboptimal on short paths this will be close to the average length, since most of the length comes from the longest paths.
- Suboptimality: This is the average suboptimality of each path. If an entry is optimal on long paths and highly suboptimal on short paths this measure will be large, since most paths are short paths.
- Num. Solved: This is the total number of problems solved out of $347,868 * 5 = 1,739,340$ total.
- Invalid: This is the number of invalid solutions returned - paths that are not well formed (e.g. adjacent locations in the returned path are not adjacent on the map).
- Unsolved: This is the number of problems that were not solved, returning no solution.
- RAM (before): This is the memory usage in MB after loading the pre-computed data.
- RAM (after): This is the memory usage in MB after running the full problem set. This includes the memory used to store the results, so this measure is artificially inflated for all entries.
- Storage: This is the disk space used for all the pre-computed storage.
- Precomputation time: This is the time (in minutes) required for the full pre-computation. Entries that perform parallel pre-computation are marked with a † in the results table in the next section.

Initial GPPC competitions limited the RAM and pre-computation times, but these limitations have been lifted to allow a more diverse set of entries at the cost of more significant computational requirements during the competition.

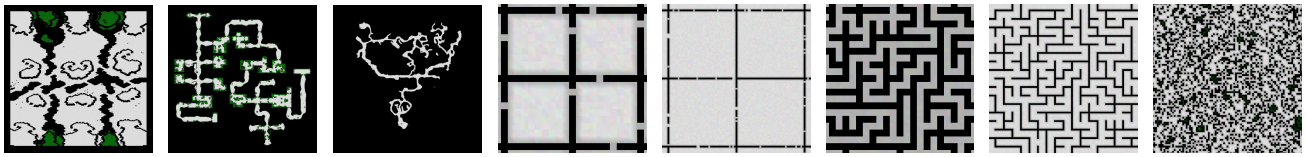


Figure 1: Sample maps from the competition.

There were 14 programs entered in the 2014 competition by 6 teams.[1] These entries are described in detail in the following sections, but there are several high-level trends in the entries. First, we see significant code re-use by new entries. This has lowered the bar for participation, allowing entrants to start with the best previous entries. The availability of past entries also facilitates experimental comparison outside of the competition. Next, we can categorize entries into several high-level approaches. These include:

- **Faster A* implementations.** In the past this included work such as pseudo-priority queues. This year there were two entries in this category, Relaxed A* (RA*) and a bucketed open list version of A*.
- **Grid Exploitation.** Several entries specifically work to exploit the structure present in grids. This includes entries based on Jump Point Search (Harabor and Grastien 2011) and Subgoal Graphs (Uras and Koenig 2014). These techniques are closely related, but a full description of their relationship is beyond the scope of this writeup.
- **Compressed All Pairs Shortest Paths.** Compressed path databases were the first approach to compressing the all-pairs shortest path data (Botea 2012). This year there are two new entries based on single-row compression (SRC).
- **Other approaches.** The subgoal approach contains some ideas from contraction hierarchies (Geisberger et al. 2008); this year we also have a contraction hierarchy entry. Past years' competitions also saw several suboptimal approaches.

2014 Competition Results

The full results of the competition can be found in Table 2. Entries in **bold** are those which are non-dominated by other entries, forming a Pareto optimal frontier. That is, they have better performance in at least one metric than all other entries. The bolded metrics for these entries are the metics across which they are non-dominated. After looking at general trends we will return to study the dominated entries.

We note the following trends in the data:

- It is difficult to devise better open list structures and guarantee correctness. The RA* did this successfully. The 2012 Pseudo-Priority Queue (PPQ) approach had a very small number of problems (23) that it could not solve. Otherwise it is significantly faster than RA* and has lower suboptimality. The A* Bucket entry had similar issues, but this entry was solicited relatively late from the organizer, and thus the participant had relatively little time to test the entry before submission. We expect to see a significantly improved entry next year.
- The subgoal approaches have steadily improved since 2012 and use relatively little RAM.
- The performance of the JPS entries has increased by over a factor of 10 over the span of the competition, and by an even larger factor when you consider the non-preprocessed version of the algorithm.
- Contraction hierarchies (CH), which have been successful in road networks, now have very good performance on game maps as well.[2]
- Pre-computed direction heuristics (PDH) (Parra, de Reyna, and Linares López 2012) were one of the few previous entries to take advantage of the incremental computation. This years' single-row compression (SRC) entry is both faster and optimal, but uses more memory.
- The fastest entry for finding full paths is still the 2013 Tree Cache entry by Ken Anderson; new entries have focused primarily on optimal paths.
- Although strictly speaking the BLJPS and NSubgoal entries use the least RAM, many entries use similar amounts of RAM, and the differences are not necessarily meaningful in practice.

Due to space concerns, we do not break down this data down further here (e.g. by map type or by path length). The full data is available for others who are interested in such analysis. We do, however, note here why each entry is dominated or non-dominated, beginning with past entries.

- Past Entries
  - The 2012 JPS entry is non-dominated because no other optimal entries do not use pre-computation.
  - The 2012 JPS+ entry is weakly dominated by the 2014 JPS+ entry. (Both entries are optimal, so they can never be fully dominated.)
  - The PPQ entry (Guivant, Seton, and Whitty 2012) would be non-dominated, except that it didn't solve 23 problems.

[1] Code for these entries as well as detailed results can be found on http://movingai.com/. Previous entries were stored on Google code, but this service, at the time of this writing, is about to shut down.

[2] The organizer worked with Robert Geisberger to use CHs on grid maps in 2010, but the results were not nearly as good.

Table 2: 2014 GPPC Results with several previous results for comparison.

2014 Entries													
Entry	Total (sec)	Avg. (ms)	20 Step (ms)	Max. Segment	Avg. Len.	Avg. Sub-Opt	Num. Solved	Num. Invalid	Num. Unsolved	RAM (before)	RAM (after)	Storage	Pre-cmpt. (min.)
RA*	492,223.7	282.995	282.995	282.995	2248	1.0580	1739340	0	0	**32.05**	58.91	**0MB**	0.0
BLJPS	25,139.4	**14.453**	14.453	14.453	2095	**1.0000**	1739340	0	0	13.58	42.59	**20MB**	0.2
JPS+	13,449.1	7.732	7.732	7.732	2095	1.0000	1739340	0	0	147.03	175.19	947MB	1.0
BLJPS2	12,947.4	**7.444**	7.444	7.444	2095	**1.0000**	1739340	0	0	13.96	42.68	**47MB**	0.2
RA-Subgoal	2,936.6	**1.688**	1.688	1.688	2142	1.0651	1739340	0	0	16.15	43.07	**264MB**	0.2
JPS+Bucket	2,811.6	1.616	1.616	1.616	2095	1.0000	1739340	0	0	379.32	407.46	947MB	1.0
BLJPS_Sub	2,731.8	1.571	1.571	1.571	2095	1.0000	1739340	0	0	19.82	48.00	524MB	2.7
NSubgoal	1,345.2	**0.773**	0.773	0.773	2095	**1.0000**	1739340	0	0	16.29	42.54	**293MB**	2.6
CH	630.4	**0.362**	0.362	0.362	2095	**1.0000**	1739340	0	0	44.66	72.04	**2.4GB**	968.8
SRC-dfs-i	329.6	0.189	**0.004**	**0.001**	2095	1.0000	1739340	0	0	246.92	356.14	28GB	†11649.5
SRC-dfs	251.7	**0.145**	0.145	0.145	2095	**1.0000**	1739340	0	0	246.92	274.16	28GB	†11649.5
A* Bucket	59,232.8	36.815	36.815	36.815	2206	1.0001	1608910	80830	49600	3577.97	3605.79	0MB	0.1
SRC-cut-i	358.1	0.208	0.004	0.001	2107	1.0000	1725440	0	13900	431.09	540.94	52GB	†12330.8
SRC-cut	276.5	0.160	0.160	0.160	2107	1.0000	1725440	0	13900	431.09	458.49	52GB	†12330.8
Past Entries													
JPS (2012)	108,749.9	62.524	62.524	62.524	2095	**1.0000**	1739340	0	0	252.18	278.97	**0MB**	0.0
JPS+ (2012)	36,307.5	20.874	20.874	20.874	2095	1.0000	1739340	0	0	356.28	383.05	3.0G	74.0
PPQ (2012)	28,930.3	16.634	16.634	16.634	2098	1.0033	1739225	115	0	47.87	74.05	0MB	0.0
Block (2012)	23,104.1	13.283	13.283	13.283	2413	1.1557	1739340	0	0	65.75	103.19	0MB	0.0
Subgoal (2012)	1,944.0	1.118	1.118	1.118	2095	1.0000	1739340	0	0	17.33	43.78	554MB	15.0
Subgoal (2013)	1,875.2	1.078	1.078	1.078	2095	1.0000	1739340	0	0	18.50	44.96	703MB	3.5
PDH (2012)	255.7	0.147	**0.008**	0.007	2259	1.1379	1739340	0	0	20.21	53.53	**649MB**	13.0
Tree (2013)	50.9	**0.029**	0.029	0.029	2564	2.1657	1739340	0	0	16.58	48.80	568MB	0.5

  - Block A* (Block) (Yap et al. 2011) is dominated by BLJPS except for storage. But, Block A* computes a 60MB table at runtime without saving it, as can be seen by RAM usage, so we still consider this entry to be dominated.
  - The 2012 and 2013 Subgoal entries are dominated by the NLevelSubgoal entry from 2014.
  - The PDH entry is non-dominated because there is only one entry with faster 20-step time, but it uses more storage.
  - The Tree entry is non-dominated because no entry finds full paths faster than this entry.
- 2014 Entries
  - The RA* entry uses the least amount of RAM for any entry that does not perform precomputation, so it is not dominated.
  - BLJPS is dominated by BLJPS2 except for precomputed storage. It is the fastest optimal entry using 20MB or less of storage.
  - JPS+ is dominated by BLJPS2, although the difference is within the 99% confidence bounds described below.
  - BLJPS2 is non-dominated because it is the fastest optimal entry using 47MB or less of storage.
  - RA-Subgoal is non-dominated because it is the fastest entry using 264MB or less of storage.
  - JPS+ Bucket is weakly dominated by NSubgoal with the exception of pre-processing time. But, these differences are not necessarily meaningful.
  - BLJPS_Sub is weakly dominated by NSubgoal with the exception of pre-processing time. But, these differences are not necessarily meaningful.
  - NSubgoal is non-dominated because it is the fastest optimal algorithm that uses 293MB or less of storage.
  - CH is non-dominated because it is the fastest optimal algorithm using 2.4GB or less of storage.
  - SRC-dfs-i is non-dominated because it is the fastest incremental algorithm (20 step and max segment time).
  - SRC-dfs is non-dominated because it is the fastest algorithm returning complete, optimal paths.
  - A* Bucket would be non-dominated if it solved all problems with better suboptimality or speed than all other entries that do not perform pre-computation. (This entry didn't receive enough time for testing because it was a last-minute solicitation from the competition organizer.)
  - The SRC-cut entries are dominated by the SRC-dfs entries.

These results show that there are many algorithms on the Pareto frontier.

In Table 3 we show the 95% confidence intervals for the timing of each of the 2014 entries. These values are based on the fact that each entry was run 5 times on each problem. Thus, we use a t-test with five samples to compute the confidence interval. Although the number of samples is small, the confidence intervals are also small because we are running so many problems (347,868) in each of the five samples. All entries can be distinguished with 95% confidence. The clos-

est results are between the JPS+ and BLJPS2 entries; these entries cannot be distinguished with 99% confidence.

2014 Competition Entries

In the following text, the authors of each entry describe their own approach in detail. The sections are ordered according to the overlapping content in each section.

Subgoal Graphs

The authors of this entry are Tansel Uras and Sven Koenig from USC. They submitted the entry NSubgoal in Table 2.

This section describes two variants of Subgoal Graphs. Simple Subgoal Graphs are constructed from grids by placing subgoals at the convex corners of obstacles and connecting them. They are analogous to visibility graphs for continuous terrain but have fewer edges and can be used to quickly find shortest paths on grids. The vertices of a Simple Subgoal Graph can be partitioned into different levels to create N-Level Subgoal Graphs (this year's entry), which can be used to find shortest paths on grids even more quickly by ignoring subgoals that are not relevant to the search, which significantly reduces the size of the graph being searched.

Simple Subgoal Graphs Simple Subgoal Graphs (SSGs) (Uras, Koenig, and Hernández 2013) are constructed from grids by placing *subgoals* at the convex corners of obstacles and connecting pairs of subgoals that are *direct-h-reachable*. Definition 1 formally defines these concepts. Vertices are put at the centers of unblocked cells. The length of an edge is equal to the Octile distance between the vertices it connects. Figure 3 shows an example of an SSG. Observe that subgoals C and E are h-reachable but not direct-h-reachable (due to subgoal D), so there is no edge connecting them.

Definition 1. *A cell s is a* subgoal *if and only if s is unblocked, s has a blocked diagonal neighbor t, and the two cells that are neighbors of both s and t are unblocked. Two cells s and t are* h-reachable *if and only if the length of a shortest grid path between them is equal to the Octile distance between them. They are* direct-h-reachable *if and only if they are h-reachable and none of the shortest paths between them pass through a subgoal.*

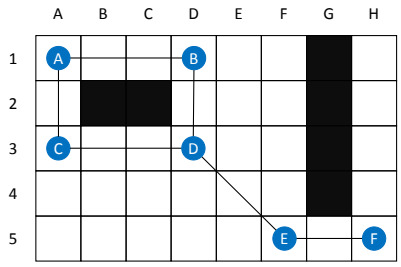


Figure 3: A Simple Subgoal Graph.

To find shortest grid paths using SSGs, one connects the given start and goal vertices s and g to their respective direct-h-reachable subgoals and searches this graph with A* to find a sequence of direct-h-reachable subgoals connecting s and g, called a *shortest high-level path*. One can then determine a shortest grid path between consecutive subgoals to find a shortest grid path between s and g. For instance, if we were to use the SSG in Figure 3 to find a shortest grid path between B1 and H3, we would connect B1 to subgoals A and B, H3 to subgoal F, and search this graph to find the shortest high-level path B1-D1-D3-F5-H5-H3. Following this high-level path on the grid, we obtain the shortest grid path B1-C1-D1-D2-D3-E4-F5-G5-H5-H4-H3.

Identifying direct-h-reachable subgoals from a given cell can be done efficiently with a dynamic programming algorithm that uses precomputed clearance values. Using this algorithm, SSGs can be constructed within milliseconds and the start and goal vertices can be connected to the SSGs quickly before a search.

N-Level Subgoal Graphs N-Level Subgoal Graphs (Uras and Koenig 2014) are constructed from SSGs by creating a hierarchy among its vertices. This hierarchy is very similar to Contraction Hierarchies (Geisberger et al. 2008; Dibbelt, Strasser, and Wagner 2014a) described later in this paper, except that N-Level Graphs can place multiple vertices at the same level of the hierarchy and only add new edges between h-reachable subgoals. The construction process assigns a level of 1 to every subgoal and then repeatedly partitions the highest-level subgoals into *global* and *local* subgoals and increases the level of global subgoals by one (any subgoal that is not at the highest level in the beginning of partitioning is ignored during partitioning). This process continues until the highest-level subgoals can no longer be partitioned or the number of levels in the graph reaches a user-defined limit. When partitioning the highest level subgoals into global and local subgoals, the partitioning has to satisfy the following property: If any subset of local subgoals is removed from the graph, the lengths of the shortest paths between the remaining local and global subgoals must remain the same. Figure 4 shows a Two-Level Subgoal Graph constructed from the SSG in Figure 3 (by adding an edge between subgoals D and F).

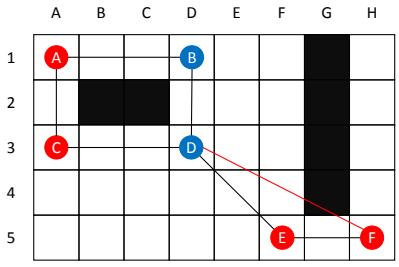


Figure 4: A Two-Level Subgoal Graph. Level 1 subgoals are shown in red, and level 2 subgoals are shown in blue.

To find shortest paths using N-Level Subgoal Graphs, one first connects the given start and goal vertices s and g to their respective direct-h-reachable subgoals, identifies all subgoals reachable from s and g via *ascending edges* (edges from a subgoal to a higher-level subgoal, s and g are assumed to have level 0 if they are not subgoals) and searches

Table 3: 2014 GPPC timing results with 95% confidence intervals.

	Run		Path		20 Step		Segment	
Entry	Avg	95%	Avg	95%	Avg	95%	Avg	95%
RA*	98,444.7	657.7	282.995	1.890	282.995	1.890	282.995	1.890
BLJPS	5,027.9	5.1	14.453	0.015	14.453	0.015	14.453	0.015
JPS+	2,689.8	68.5	7.732	0.197	7.732	0.197	7.732	0.197
BLJPS2	2,589.5	0.9	7.444	0.003	7.444	0.003	7.444	0.003
RA-Subgoal	587.3	0.4	1.688	0.001	1.688	0.001	1.688	0.001
JPS+Bucket	562.3	1.3	1.616	0.004	1.616	0.004	1.616	0.004
BLJPS_Sub	546.4	0.7	1.571	0.002	1.571	0.002	1.571	0.002
NSubgoal	269.0	0.8	0.773	0.002	0.773	0.002	0.773	0.002
CH	126.1	4.8	0.362	0.014	0.362	0.014	0.362	0.014
SRC-dfs-i	65.9	0.1	0.189	0.000	0.004	0.000	0.001	0.000
SRC-dfs	50.3	0.2	0.145	0.000	0.145	0.000	0.145	0.000
A* Bucket	11,846.6	110.1	36.815	0.342	36.815	0.342	36.815	0.342
SRC-cut-i	71.6	0.3	0.208	0.001	0.004	0.000	0.001	0.000
SRC-cut	55.3	0.2	0.160	0.001	0.160	0.001	0.160	0.001

the graph consisting of those subgoals and all highest-level subgoals (and the edges between them), thus ignoring other subgoals during search. For instance, if one were to use the Two-Level Subgoal Graph in Figure 4 to find a path between B2 and H3, the graph searched would include the subgoals A, B, D, and F but not C and E.

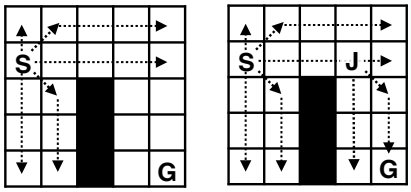
The ordering in which vertices are considered during the construction process determines the hierarchy obtained. As Contraction Hierarchies show, good node contraction orderings can speed up path planning significantly, which is why Contraction Hierarchy approaches often spend a significant amount of preprocessing time on finding good node contraction orderings. It is future work to consider this effect in the context of subgoal hierarchies, whose GPPC entries used an arbitrary node ordering.

Jump Point Search (JPS)

As several entries are built upon Jump Point Search (JPS) (Harabor and Grastien 2011; 2014), we provide a basic description of the algorithm as an introduction to these entries.

JPS leverages the structure of grids to avoid reaching many states via alternate/redundant paths. The core idea of JPS is to order paths such that all diagonal moves are taken first, followed by horizontal/vertical moves. This creates a canonical ordering over all paths, ensuring that there is only one path to each goal. However, this idea is not sufficient on its own, because obstacles may block the natural canonical path to a given goal. Thus, jump points are introduced. Jump points are locations in the map where the canonical ordering is partially reset in order to allow the search to move around obstacles. We illustrate this in Figure 5. In Figure 5(a) we show the states reachable according to the canonical ordering of states starting at S. In Figure 5(b) we add a jump point, labeled J, where the canonical ordering is reset, allowing the search to proceed down and to the right, reaching the goal. The new neighbors of J are called *forced* neighbors.

The jump points depend on the obstacles in the map and can be used at runtime or pre-computed beforehand. JPS is



(a) (b)

Figure 5: Basic JPS example.

efficient not only because it limits the number of symmetric paths, but also because it reduces the number of nodes stored in the open list. Instead of placing each successor of a node on the open list, only jump points need to be put on the open list. The following two entries build upon JPS to improve its performance.

BLJPS

The authors of this entry are Jason Traish and James Tulip from Charles Sturt University. They submitted the entries BLJPS, BLJPS2 and BLJPS_Sub in Table 2.

Boundary Lookup Jump Point Search (BLJPS) is an extension of Jump Point Search (JPS). BLJPS preprocesses the boundaries on a map to speed up jump point detection. Boundaries are used to identify jump points that exists along the horizontal or vertical axis without iteratively checking numerous cells.

Normal JPS behavior allows diagonal moves when the relevant cardinal directions are blocked. However, for the purposes of this competition, such moves are forbidden. In the submitted code, toggling between these behaviors is controlled by the flag variable DIAG_UNBLOCKED.

The following example uses Figure 6 to demonstrate how JPS moves from P1 to P4. Each cardinal direction check is displayed as a dotted line. The search starts at P1 by checking all 8 directions (cardinal plus diagonal) for Jump Points. In this case, only the East, North and NE directions are ac-

cessible. JPS iterates over open locations along a cardinal direction until it reaches a blocked location, at which point it terminates without returning a jump point. In this case, both East and North fail to return a jump point. The algorithm then expands diagonally in the NE direction to location (B8). It then repeats the iterative cardinal direction jump point checks relative to NE (North and East). Again, it fails to find a jump point along either axis, and the process is repeated until the diagonal expansion reaches a boundary or a Jump Point is found in one of the relative cardinal directions. In this case, the NE expansion continues to the location P2, at which point the Eastern cardinal direction check returns a potential Jump Point at P3, identified by the presence of a forced neighbor. P2 is then added to the open list with the direction NE. The discovery or not of a Jump Point in the NE direction terminates the search from P1, since only N, E, and NE directions were accessible.

The top node within the open list is then popped, returning P2. Natural neighbor directions for a NE expansion (N, E, NE) are then searched. The North and NE searches find no further jump points, but the East direction has P3 as a potential Jump Point. P3 is evaluated as an actual Jump Point in a manner similar to P2. However, the SE direction is also searched as a result of the forced neighbor at P4. Stepping in the SE direction identifies that the goal P4 has been reached. This confirms P3 as a Jump Point and completes the path (P1, P2, P3, P4).

Table 4 shows the boundary lookup table corresponding to Figure 6. The values in Table 4 record boundaries where cells toggle between blocked and open states, starting from the western (or northern) boundaries. Rows such as 1 through 5 have no obstructions resulting in the horizontal boundary lookup recording the map width (N). Rows 6 through 9 however have a boundary at K and thus record this as the first entry. The boundary then reopens on cell L and again is blocked by the width of the map (N).

BLJPS identifies jump points in eastern/western directions when it has a boundary that is further in the given direction than the reopen value for the rows above and below it. For example (A5) to the East has a boundary of N which is less than the reopen value of row below it (Row 6). Row 6's boundary is K and reopens to the east at L which is less then N, resulting in a jump point found at (L-1=K row 5, K5) or P3. If the direction is changed to West moving from (M5) the boundary is off the map at -A. The row 6 that is beneath 5 hits K as the western boundary and reopens on (K-1=L) which is not as far in the western direction as the -A boundary.

BLJPS2 optimizes this approach by storing the jump points for the four cardinal directions within separate lists. This reduces the number of table lookups from three to one for axis each check.

BLJPS SubGoal is a hybrid approach between BLJPS and SubGoal. It uses the same high level graph pruning approach as SubGoal, but substitutes low level subgoals with Jump Points in order to exploit the directionality of JPS. However, attaching the goal node to the low level SubGoal graph proved computationally expensive, and no improvement over the basic SubGoal algorithm was noted. SubGoal

Table 4: Boundary Lookup Table - Used to determine the nearest obstacle boundary along either the horizontal or vertical axis.

Horizontal Boundary Lookup	Vertical Boundary Lookup
Cells 1 to 5: (N)	Cells A to J: (10)
Cells 6 to 9: (K,L,N)	Cell K: (6)
	Cell L, M, N: (10)

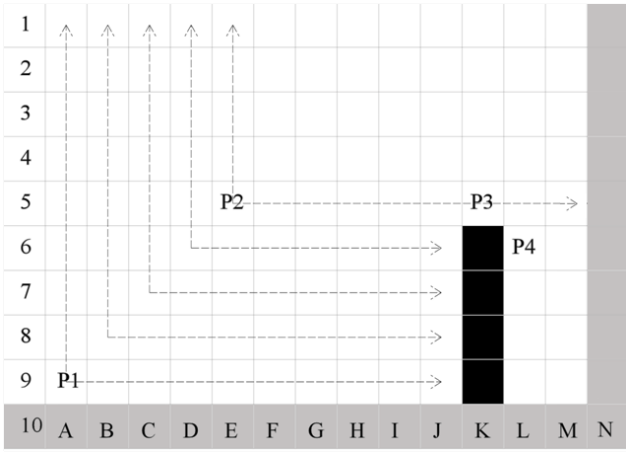


Figure 6: Boundary lookup example on a 13x9 uniform grid. Blacked cells K6-K9 are obstacle boundaries. P4 is a forced neighbor of P3.

was retrieved from 2013 entries of the Grid-Based Path Planning Competition.

Optimized JPS+

The author of this entry is Steve Rabin. Submitted entries include JPS+, JPS+Bucket, and A* Bucket.

JPS+ precomputes 8 values per node (one per edge), which helps guide the search at runtime to predetermined jump point nodes. Significant node pruning is realized through a combination of search via a canonical ordering and jumping over nodes on the way to pivotal jump point nodes. This has the effect of speeding up the search by examining fewer nodes and consequently maintaining a reduced set of nodes on the open list.

Optimized JPS+ was programmed using a number of best practices, each contributing a small amount to the overall speed. The node data maintained for the search was preallocated in a 2D array and a search iteration count stored in each node avoided the need to reset node data before each search. All costs were performed using fixed-point math and the heuristic was calculated with the octile heuristic: between two points we compute the maximum and minimum distances in each cardinal direction. The octile distance is then $(\text{minDiff} * (\sqrt{2} - 1.0) + \text{maxDiff})$. Additionally, if a node was already on the open list and its cost needed to be updated, the heuristic was extracted, thus avoiding having to recalculate it. Finally, the open list used a heap priority queue augmented with a hash table to speed up the

DecreaseKey operation that occurs when A* finds a shorter path to an existing node in the search.

The Bucket version of optimized JPS+ exchanged the heap priority queue for a priority queue implemented using buckets, indexed by cost, containing a stack within each bucket. The required number of buckets is 10x the largest search cost with a node's bucket index equalling 10 times the final cost, resulting in a 0.1 cost variation within each bucket. A pointer kept track of the cheapest non-empty bucket, which was updated on adds and removals as necessary. A search iteration count was used inside each bucket to implement a lazy reset with each new search.

Two significant optimizations were not implemented in time for the competition, but each have been shown to speed up JPS+ search even further. Using a fast stack optimization is very helpful, as implemented in the 2013 contest entry of Subgoal. This optimization puts nodes on a stack instead of in the open list if they are equal to the cost of the parent. Nodes are then removed off the fast stack before checking the open list. Note that this optimization is only valid on grid search spaces using the octile heuristic. Additionally, a function pointer lookup table based on the wall pattern and parent direction has been shown to speed up searches by eliminating conditionals (2,048 entries pointing at 48 unique functions). Lastly, as a memory optimization, the number of buckets can be reduced by rebasing all search costs by the original octile heuristic cost of the start node.

For comparison purposes, A* Bucket is an A*-based entry using buckets for the open list in a similar fashion to the JPS+.

Relaxed A*

This entry was written by a research team at `http://www.iroboapp.org/`. Due to non-scientific concerns, the author of this work were unable to participate in this paper; this description is written by the competition organizer.

The researchers in this group submitted the entries labelled RA* and RA-Subgoal in Table 2. Their entry relaxes the behavior of A* in three ways. First, it doesn't allow the g-cost of a node to be changed once it has been set the first time. Second, since nodes only have their g-cost updated once, the open list doesn't have to be explicitly checked for membership when opening a node with A*. All nodes on the open will have infinite g-costs. Finally, the entry doesn't maintain parent pointers, but assumes that the parent of a node is the one with the lowest g-cost.

This approach is applied first to a regular A* search, and then to the A* search that is part of subgoal graphs (Uras, Koenig, and Hernández 2013). The subgoal code is built from the SubgoalGraph-Fast code from the 2013 GPPC competition.

Contraction Hierarchies (CH)

The author of this entry is Ben Strasser from Karlsruhe Institute of Technology, labeled CH in Table 2.

A CH is a speedup technique to compute shortest paths in general weighted graphs. The technique works in two phases: A *preprocessing* and a *query phase*. The preprocessing phase is slow and augments the graph with auxiliary data. The endpoints of the desired path are the input of the query phase. The query phase can make use of the auxiliary data and should run very fast. The technique was introduced in (Geisberger et al. 2012) and has been the focus of many research papers since then. The technique has been evaluated very extensively on road graphs in various independent studies. However, there have been questions about whether the technique is feasible on graphs originating from game grids or whether the technique needs tweaking. The sole goal of this submission is to establish this. We have not optimized the submission for game maps. In fact the code is taken from the implementation of (Dibbelt, Strasser, and Wagner 2014b) and mostly wrapped. It should be possible to tune the code. For example the preprocessing code does not exploit that game graphs are undirected. Doing so should be straight-forward and should yield a factor of 2 in terms of preprocessing running time.

The core idea of the algorithm is easy but needs some notation. Denote by G the input graph, which for this competition is undirected. The auxiliary data consists of a directed acyclic *search graph* G' with the same node set than G that fulfills the *cover-property*: For each pair of source node s and target node t, there exists a node m, such that for each shortest st-path P in G, there exists a sm-path P'_u and a tm-path P'_d in G' such that the length of P is the same as the sum of the lengths of P'_u and P'_d. By convention we say that arcs in G' point from the bottom to the top. We refer to P'_u as the *upward* path and to P'_d as the *downward* path. We refer to the pair of P'_u and P'_d as an *up-down* path.

Think of the top nodes as important nodes that cover many shortest paths. These could for example be bridges in road graphs. The bottom nodes are very unimportant. Think of them as dead-ends. You can think of an up-down path as a coarsened version of a shortest path in G. Coarsened means that all the edges between two top nodes, i.e. bridges, are missing. The idea of the algorithm consists of searching for P'_u and P'_d, i.e. an up-down path, instead of P. It does so using two instances of Dijkstra's algorithm that operate on G'. The first instance searches starting from s and the second instance from t. The node m will be found by both searches. The first instance finds P'_u and the second finds P'_d and from there the path in G can be reconstructed. We refer to the subgraph of G' reachable from a node x as the *search space* of x. The key ingredient is to find a graph G' such that all search spaces are small.

To construct G' we need the concept of (weighted) node-contraction. This is the operation that has given CHs their name. A node contraction consists of removing a node x from a graph H and inserting an edge between two neighbors y and z if the path $y{\rightarrow}x{\rightarrow}z$ is the only shortest yz-path. Testing whether $y{\rightarrow}x{\rightarrow}z$ has this property is done by running Dijkstra's algorithm on $H\backslash\{x\}$ to check whether the shortest yz-path in $H\backslash\{x\}$ is longer than $y{\rightarrow}x{\rightarrow}z$. This operation is called *witness search*. The idea is to remove x from H while maintaining all shortest path distances in $H\backslash\{x\}$. G' is constructed from G by iteratively contracting unimportant nodes along a *contraction order* $o_1 \dots o_n$. Denote by G_i

the graph that remains after the first $i-1$ nodes have been contracted, i.e., $G_1 = G$ and G_i is obtained by contracting o_{i-1} in G_{i-1}. While enumerating these G_i the algorithm constructs G' as following: The outgoing arcs of o_i are towards the neighbors of o_i in G_i. This construction fulfills the cover-property as was shown in (Geisberger et al. 2012).

The remaining key question is how to determine a good node contraction order. This is by far the slowest part of the preprocessing. Several approaches exist in the literature. The original CH-paper (Geisberger et al. 2012) uses a bottom-up approach that greedily determines the most unimportant node during the CH contraction. In (Abraham et al. 2012) a top-down approach was introduced. The idea is that the most important node is the node that lies on the most non-covered shortest paths. (A path is covered if it contains a node already put into the order.) Bottom-up approaches work extremely well in practice on road-like graphs but have the elegance of a cooking recipe: You throw a mixture of heuristics at the problem and then in the end it somehow works out. To the best of our knowledge nobody really knows why one recipe is superior to some other recipe. Top-down approaches are more elegant but in their basic variant slower. In (Delling et al. 2014) a fast sampling based top-down approach was introduced. Both the bottom-up and top-down approaches compute orders that depend on the edge weights of G. Changing the weights requires computing a new order. In (Dibbelt, Strasser, and Wagner 2014b; Bauer et al. 2013) a third weight-independent construction based on balanced graph separators was introduced and shown to come extremely close in terms of query performance to weight-dependent orders. This weight-independent construction is tightly coupled with the theory of tree-decompositions.

The GPPC submission uses a comparatively slow but high-quality bottom-up ordering approach. The core idea is to interweave the construction of G' and the contraction order. The algorithm determines for every node x in G_i an *importance* value $I_i(x)$. It then sets $o_i = \min\arg I_i(x)$. In the following we will drop the i-index and from I_i to avoid unnecessary notational clutter. To avoid recomputing all $I(x)$ in each iteration a priority queue is used that orders all nodes by their current importance. Contracting a node only modifies the importances of neighboring nodes. These values are updated in each iteration and the algorithm continues by contracting the node with a minimum $I(x)$. The definition of $I(x)$ is a cooking recipe. We start by defining all the ingredients. We denote by $L(x)$ a value that approximates the level of vertex x in G'. Initially all $L(x)$ are 0. If x is contracted then for every incident edge $\{x, y\}$ we perform $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$. We further store for every arc a a hop length $h(a)$. This is the number of arcs that the shortcut represents if fully unpacked. Denote by $D(x)$ the set of arcs removed if x was contracted and by $A(x)$ the set of arcs that would be inserted. The sets $A(x)$ and $D(x)$ are computed by simulating the node contraction. We set:

$$I(x) = L(x) + \frac{|A(x)|}{|D(x)|} + \frac{\sum_{a \in A(x)} h(a)}{\sum_{a \in D(x)} h(a)}$$

This formula originates from (Abraham et al. 2012). Note that the original CH variant (Geisberger et al. 2012) contains significantly more heuristic aspects that aim at reducing preprocessing time. For example their variant aborts the witness searches if Dijkstra's algorithm runs for too long and it does not update all $I(x)$ but use a technique called *lazy-pop*. Our GPPC submission does none of this.

To illustrate the performance of CHs we ran an experiment on the StarCraft IceFloes map. It has 91,123 cells which translate a graph G with 91,123 nodes and 347,624 edges with weights 1 or $\sqrt{2}$. Determining the order needs a bit less than 2 minutes. The search graph G' contains 733,114 arcs. On average the search space of a random uniform node contains 680 nodes and 6,677 arcs. The search graph G' has 48 levels. This CH performance is very good. It is only very rarely possible to get less than two search graph arcs per input edge while maintaining search spaces of reasonable size. However, we believe that by exploiting the map-structure of the input instances significant improvements can still be made. For example the encoding of the input map as two-dimensional bitmap is far more memory efficient than any equivalent general purpose graph representation. Leveraging this observation could lead to a significantly lower memory footprint.

Single Row Compression (SRC)

The authors of this entry are Ben Strasser, Adi Botea, and Daniel Harabor, corresponding to the entries SRC-dfs, SRC-dfs-i, SRC-cut, SRC-cut-i in Table 2.

SRC is a speedup technique to compute shortest paths in general weighted graphs of bounded degree. The technique was introduced in (Strasser, Harabor, and Botea 2014) and its theoretical complexity studied in (Botea, Strasser, and Harabor 2015). It works in two phases: A *preprocessing* and a *query* phase. The preprocessing phase is slow and augments the graph with auxiliary data. The endpoints of the desired path are the input of the query phase. The query phase can make use of the auxiliary data and must run very fast.

A very elementary technique that uses this setup consists of computing a large matrix A of first moves. A *first move* from the source node s to the target node t is the first edge of a shortest path from s to t. Each entry a_{ij} of the matrix A corresponds to the first edge of a shortest path from the i-th node to the j-th node. By convention we say that the i-th row contains all first moves from the i-th node to all other nodes. Similarly the j-th column contains all first move towards the j-th node. The query phase of this setup is extremely fast and consists most of the time of a single random memory access. The downside of this approach is that the matrix needs space quadratic in the node count and this is prohibitive for large graphs.

SRC mitigates this space problem by exploiting a very simple observation: All edges in a row must be outgoing edges of a single node. If we assume that the graph is of bounded degree then many edges must be repeated. SRC exploits this fact by compressing each row individually using a *run-length encoding* (RLE). If done properly, query running times can be answered in time logarithmic in the length of a compressed row. Unfortunately, it is possible that rows do not compress well using RLE. SRC therefore first per-

mutes the nodes in such a way that similar first-moves tend to appear adjacent in a row. Unfortunately, computing a node order that minimizes the number of runs is NP-hard as was shown in (Botea, Strasser, and Harabor 2015). Fortunately heuristics work well in practice: In (Strasser, Harabor, and Botea 2014) the *dfs-order* and the *cut-order* were proposed. The dfs-order is extremely simple and consists of reordering the nodes using a depth-first preorder from a random root node. The cut-order is more complex and recursively partitions the graph. On most tested graphs the cut-order works slightly better than the dfs-order. However for unknown reasons on some graphs the cut-order is far inferior. This leads to a bad compression ratio. This in term leads to an integer overflow which explains why the cut-order submission fails on a few graphs. To gain an intuition for how a good order looks like consider the USA-road graph. Consider the row R of some node in the south east. If the destination is in the north west, the path will most likely always start with the same first move. The key to success is finding a node order that assigns continuous node ids to all nodes in the north west. Such an order assures that all of the north west part of the USA collapses to a single run in R.

The strength of SRC lies in computing first-moves very quickly. The main downside lies in its preprocessing time that is quadratic in the node count as it must compute A. The compressed space consumption is large compared to some other techniques but manageable. Notice that computing first moves quickly does not automatically translate into the fastest algorithm to compute full paths quickly. The reason is inherent to all techniques that compute paths an edge at a time. This includes even the elementary quadratic matrix approach. Such algorithms need to pay a random memory access per edge. An algorithm that has large parts of the paths preassembled in memory only needs to pay one random memory access per part. Because of caching effects this can be significantly faster. (A sequential memory access can be about 70 times faster than a random one on current hardware.) However, in many important scenarios the more cache friendly path computation does not pay off in the end. Consider a game unit that needs to navigate to a certain position. There are two different setups: (1) When starting its journey the unit computes the whole path and stores it somewhere. Each time the unit reaches the end of an edge it looks up the next edge, i.e., a random memory access is needed. (2) Each time a unit reaches the end of an edge it computes only the next edge. Setup (1) can look better than (2) in laboratory conditions because measured running times often only account for the time needed to compute the path. The time that the unit spends looking up the individual edges is often neglected. Further, the approach (1) needs to store a variable length path in memory. This results in addition time needed for memory management and more importantly in a non-constant memory consumption per unit. Consider the extreme case where there are $\omega(n)$ units that have stored path with $\Theta(n)$ edges (n being the number of nodes). This results in $\omega(n^2)$ memory consumption. Even setup 2 with the quadratic matrix approach uses only $\Theta(n^2)$ memory in this case.

The Table 2 contains 4 variants of our algorithm. The cut/dfs part refers to the node order used. The "-i" indicates whether the full path was extracted in one function call or whether the path was extracted edge by edge. In theory this should not make a difference. However, in practice it influences how often the timing code is called. If the path is extracted in one function call the timing code only runs before and after each path query. However, if the path is extracted edge-by-edge the timing code runs once per edge. This slightly different experimental setup results in the 0.044ms difference between the average path extraction running times of "dfs-i" and "dfs".

Looking Ahead and Conclusions

The GPPC covers a variety of grid-based pathfinding problems. Several primary competing approaches have been developed which take advantage of the grid structure and significantly improve performance over basic A* implementations. Thus, the competition has successfully documented significant improvement in grid-based pathfinding and made high-quality pathfinding implementations available for experimentation and testing.

Looking at applications of the work, the competition covers a subset of the problems that are interesting to the games industry, but doesn't address some real-world problems that must sometimes be solved when grids are being used. In particular, grids are often used in dynamic environments where the world and/or the cost of moving through the world changes regularly (Sturtevant 2013). This is because a grid representation is easy and inexpensive to update as the world changes.

The current competition doesn't allow for dynamic weights in the map or dynamic maps, and a new track designed to foster this sorts of comparisons did not receive any entries. In some sense this isn't completely surprising, as there has been many years of work on grid methods, so there was a large pool of work ready to enter into the competition. Also, there are many ways to test dynamic maps, and there may be a need for different testing for more robotic-like environments versus game environments.

But, after several years of the competition, we can see a core of main ideas that are used to improve performance, with a range of memory and pre-computation requirements. As these approaches are iteratively improved, we look forward to seeing additional techniques which fill in the gap between these entries. In particular, there is significant room for innovation for real-time and suboptimal entries.

Going forward, it is important to address how to challenge the research community to move beyond the uniform cost grid pathfinding problem and into problems that are more representative of real-world problems faced by practitioners. For instance, the first author of this paper is working with the games industry to get more maps and map types that will give meaningful data to be studied by researchers. Careful consideration will be given to how the competition can be used to encourage research into different problem types that have even wider application across a broader range of domains.

Acknowledgements

The staff at the Center for Statistics and Visualization at the University of Denver provided useful discussion and feedback regarding the statistical analysis in the paper.

References

Abraham, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F. F. 2012. Hierarchical hub labelings for shortest paths. In Epstein, L., and Ferragina, P., eds., *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, volume 7501 of *Lecture Notes in Computer Science*, 24–35. Springer.

Bauer, R.; Columbus, T.; Rutter, I.; and Wagner, D. 2013. Search-space size in contraction hierarchies. In Fomin, F. V.; Freivalds, R.; Kwiatkowska, M. Z.; and Peleg, D., eds., *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, volume 7965 of *Lecture Notes in Computer Science*, 93–104. Springer.

Borrajo, D.; Felner, A.; Korf, R. E.; Likhachev, M.; López, C. L.; Ruml, W.; and Sturtevant, N. R., eds. 2012. *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press.

Botea, A.; Strasser, B.; and Harabor, D. 2015. Complexity results for compressing optimal paths. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Botea, A. 2012. Fast, optimal pathfinding with compressed path databases. In Borrajo et al. (2012).

Delling, D.; Goldberg, A. V.; Pajor, T.; and Werneck, R. F. 2014. Robust distance queries on massive networks. In Schulz, A. S., and Wagner, D., eds., *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, 321–333. Springer.

Dibbelt, J.; Strasser, B.; and Wagner, D. 2014a. Customizable contraction hierarchies. *arXiv preprint arXiv:1402.0402*.

Dibbelt, J.; Strasser, B.; and Wagner, D. 2014b. Customizable contraction hierarchies. In Gudmundsson, J., and Katajainen, J., eds., *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, 271–282. Springer.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, 319–333.

Geisberger, R.; Sanders, P.; Schultes, D.; and Vetter, C. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46(3):388–404.

Guivant, J. E.; Seton, B.; and Whitty, M. A. 2012. 2d path planning based on dijkstra's algorithm and pseudo priority queues. In Borrajo et al. (2012).

Harabor, D. D., and Grastien, A. 2011. Online graph pruning for pathfinding on grid maps. In Burgard, W., and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press.

Harabor, D. D., and Grastien, A. 2014. Improving jump point search. In Chien, S.; Do, M. B.; Fern, A.; and Ruml, W., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. AAAI.

Parra, Á.; de Reyna, Á. T. A.; and Linares López, C. 2012. Precomputed-direction heuristics for suboptimal grid-based path-finding. In Borrajo et al. (2012).

Strasser, B.; Harabor, D.; and Botea, A. 2014. Fast first-move queries through run-length encoding. In Edelkamp, S., and Barták, R., eds., *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*. AAAI Press.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144–148.

Sturtevant, N. R. 2013. Choosing a search space representation. In *Game AI Pro: Collected Wisdom of Game AI Professionals*. CRC Press.

Uras, T., and Koenig, S. 2014. Identifying hierarchies for fast optimal search. In Brodley, C. E., and Stone, P., eds., *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 878–884. AAAI Press.

Uras, T.; Koenig, S.; and Hernández, C. 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI.

Yap, P. K. Y.; Burch, N.; Holte, R. C.; and Schaeffer, J. 2011. Abstract: Block a* and any-angle path-planning. In Borrajo, D.; Likhachev, M.; and López, C. L., eds., *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, Castell de Cardona, Barcelona, Spain, July 15.16, 2011*. AAAI Press.