§CS 520: Assignment 1
Fast Trajectory Replanning

Xiaoyang Xie
167008240

Yikun Xian
168000142

Department of Computer Science
Rutgers University, New Brunswick, NJ

09 October 2015

Abstract

Heuristic search algorithms like A* can be adapted to solving path planning problems of directed goal and unknown environment. In this report, we mainly discuss and evaluate three variants of A* algorithms, namely Repeated Forward A*, Repeated Backward A* and Adaptive A*. In the experiment, we first generate two sets of 1000 random grid-based maps, where one follows the assignment requirement containing approximate 30% obstacles and 70% roads, and the other is the set of corridor-like mazes generated by DFS with randomly expanded nodes. Then we compare three algorithms by such evaluation indices as number of expanded nodes, number of explored nodes, total cost of steps, optimal cost of steps, etc. The result shows that 1) the average number of expanded nodes per map and explored nodes per map of Adaptive A* are respectively 28.06% and 24.00% less than those of Repeated Forward A*; 2) cost of steps for all three algorithms is pretty much the same. This indicates that Adaptive A* is greatly optimized in path replanning phase, while there is no significant improvement in actual moving phase. Finally, we discuss and calculate how to optimize data structures to store states as many as possible within only 4M memory. This is the practical problem in the situation where computational resources are rare and precious.

Part 0 Setup Environment

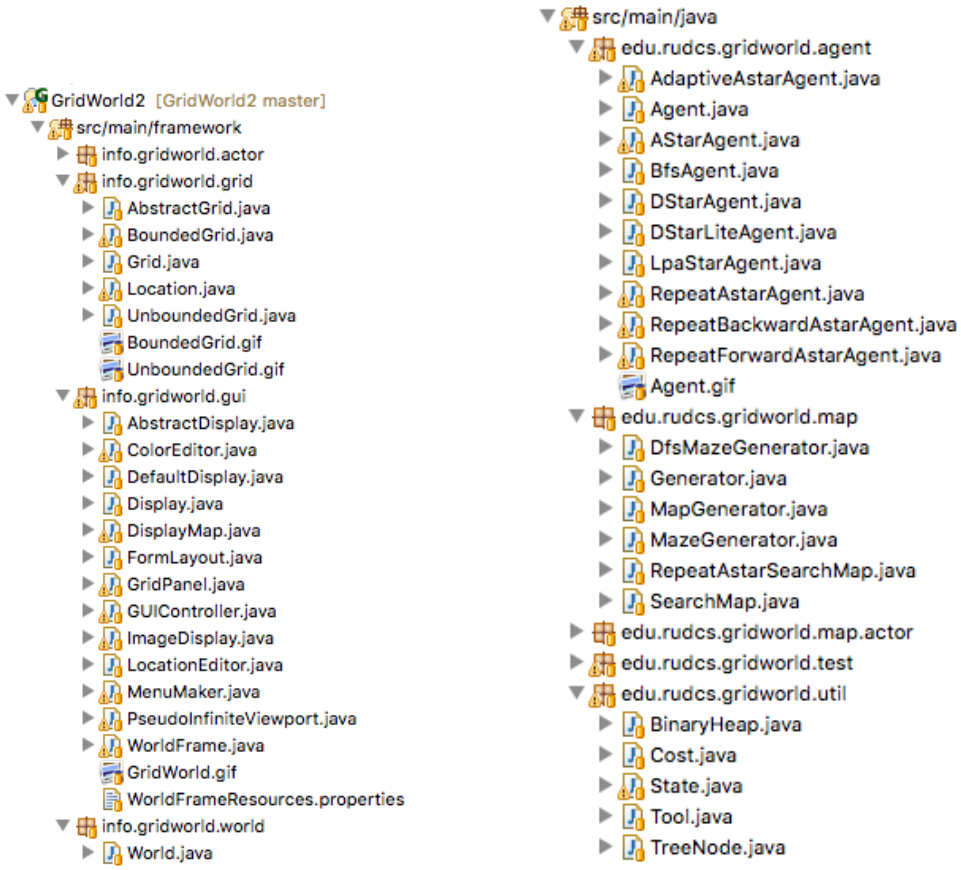
We simulate all path finding processes based on the framework of GridWorld[1], an AP case study project from collegeboard [1]. It provides graphical user interface based on Java AWT where visual objects can interact and perform customized actions in a two-dimensional grid map. In the next part, we will first illustrate original GridWorld framework and our enhancement of displaying colored path. This mainly involves the engineering work, so if you want to directly delve into algorithm analysis, please skip it.

0.1 GridWorld Architecture and Modification

The source code of original GridWorld project is placed in $src/main/framework$ folder and its structure can be divided into four parts, as shown in Figure 1a.

The *actor* package contains objects whose behavior on the map can be arbitrarily defined by rewriting *act* method in each inherited class:

[1] https://www.collegeboard.org/



(a) GridWorld structure (b) Project structure

Figure 1: Structure of GridWorld framework and path-finding project

```
@Override
public void act()
```

The *grid* package defines features on bounded and unbounded grid map as well as connections between map and actors. The *gui* package encapsulates low-level Java AWT to provide APIs for visualization and interactions of map and actors. The *world* package provides high-level integration of actors and world.

In order to visualize the presumed unblocked path and the set of nodes expanded and explored after each planning, we enhance the *GridPanel* class in the GUI package by implementing the method below:

```
private void drawColoredLocations(Graphics2D g2)
```

Meanwhile, following five abstract methods related to colored grid are added to *Grid* interface and classes like *BoundedGrid* and *UnboundedGrid* are responsible

to implement color configuration on each grid.

```
ArrayList<Location> getColoredLocations();
Color getColor(Location loc);
void putColor(Location loc, Color color);
void removeColor(Location loc);
void resetColors();
```

The source code related to assignment is placed in *src/main/java* folder, as shown in Figure 1b, which are divided into five packages. The *agent* package defines the simulated agents equipped with specific navigation algorithm for solving goal-directed path-finding problem. The *map* package provides APIs for generation of random map and corridor-like maze. The *map.actor* package defines some static object on the map like obstacles and goal. The *test* package contains simulation program for experiment and testing. The *util* package includes some self-defined data structures to be used in storing intermediate result of algorithm.

0.2 How to Run

Our project is built and packed by Gradle[2], an open-source build automation tool. To run the project, you should first install and setup Gradle environment on your computer, and install Gradle plugin for Eclipse, called "Gradle Integration for Eclipse". Then, download complete source code of the project, which is attached in Sakai or available on the Github (`https://github.com/orcax/GridWorld2`). Finally, inside Eclipse, click *File* $\rightarrow$ *Import* $\rightarrow$ *Gradle* $\rightarrow$ *Gradle Project* to import project. All runnable programs are placed in *test* package.

0.3 Maze Generation Algorithm

In the experiment, we mainly use two kinds of grid map, namely random map with discrete obstacles and random maze with consecutive obstacles. The random map is generated according to the assignment requirement, so for each map, there are about 30% obstacles and 70% roads. The start position and the goal position can be placed either randomly or on diagonal corners. Difference between these two ways lies in the absolute distance between start and goal, although, the experimental result implies that this initial distance in this type of random map has nearly no influence on the performance of three algorithms. On the other hand, the longer the distance is, the more probable it will lead to unreachable goals, so

[2]http://gradle.org/

in this case, we simply collect maps that exist a path between the start position and goal position.

Random maze is the other type of maps tested in the experiment. It is generated according to DFS by expanding neighbors randomly. The generation process is described as Algorithm 1.

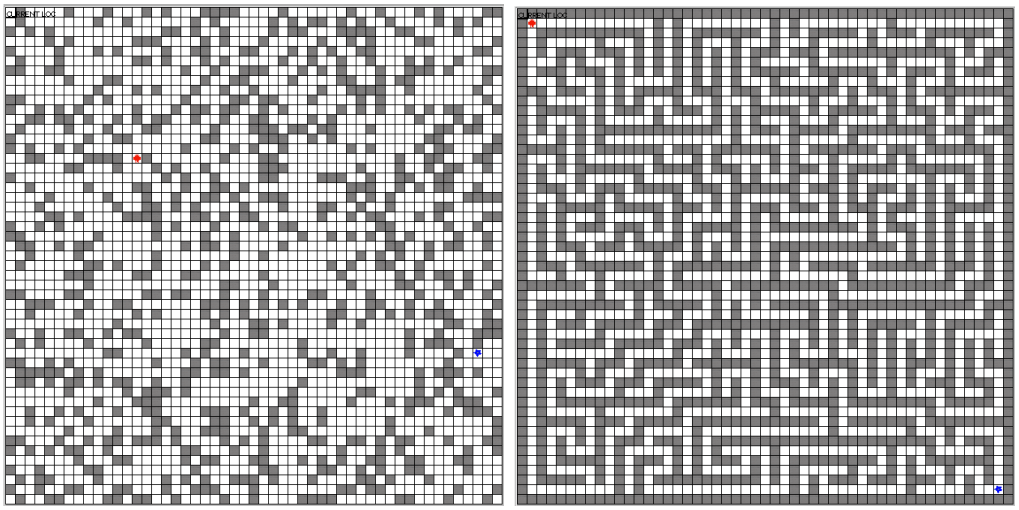
Algorithm 1 Maze Generation Algorithm by DFS

```
1: function GENERATE-BY-DFS(map)
2:     set all cells in map as WALL
3:     r = random(⌊map.rows/2⌋) * 2 + 1
4:     c = random(⌊map.cols/2⌋) * 2 + 1
5:     map[r][c] = ROAD
6:     call RANDOM-DFS(map, r, c)
7:     map[1][1] = START
8:     map[map.rows − 2][map.cols − 2] = GOAL

1: function RANDOM-DFS(map, row, col)
2:     (row′, col′)=RANDOM-NEIGHBOR(map, row, col)
3:     while (row′, col′) ≠ NULL do
4:         map[row′][col′] = ROAD
5:         map[(row + row′)/2][(col + col′)/2] = ROAD
6:         call RANDOM-DFS(map, row′, col′)
7:         (row′, col′)=RANDOM-NEIGHBOR(map, row, col)

1: function RANDOM-NEIGHBOR(map, row, col)
2:     initialize neighbors as empty list
3:     if row − 2 > 0 and map[row − 2][col] ≠ ROAD then
4:         neighbors.add((row − 2, col))
5:     if row + 2 < map.rows and map[row + 2][col] ≠ ROAD then
6:         neighbors.add((row + 2, col))
7:     if col − 2 > 0 and map[row][col − 2] ≠ ROAD then
8:         neighbors.add((row, col − 2))
9:     if col + 2 < map.cols and map[row][col + 2] ≠ ROAD then
10:        neighbors.add((row, col + 2))
       return neighbors[random(neighbors.size)]
```

In this algorithm, the map is initialized as two-dimensional array and set all cells as obstacles (WALL). Roads will be expanded randomly and paved only on the cells of odd index. To generate roads, we first begin with randomly picking a cell on the map and set it as road. Then, randomly select one of its non-road neighboring nodes (of odd index), set it and the node between them as roads. Repeat this



(a) Map with discrete obstacles (b) Corridor-like maze

Figure 2: Example of random map and random maze

procedure until there is no valid neighboring nodes. If this happens, traceback to its parent node and do the same thing again. At last, all nodes of odd index are traversed and some nodes of even index are set as road to connect two adjacent nodes. Finally, without loss of generality, we simply set top-left odd bottom-right respectively as start position and goal position.

It is easy to prove that there is only one depth first tree generated during one complete search. The connectivity of single depth first tree make the maze always reachable, namely there always exists one path from start to goal. Example of two kinds of map of size 50*50 is shown in Figure 2.

Part 1 Understanding the Methods

1.1 Problem A

Problem Explain in your report why the first move of the agent for the example search problem from Figure 3 is the east rather than then north given that then agent does not know initially which cells are blocked.

Solution In Figure 3, the start state A is at $(E, 2)$ and goal state T is at $(E, 5)$. Let's denote A_N as the north cell $(D, 2)$ of A, so $g(A_N) = 1$ and $h(A_N) = 3$, where

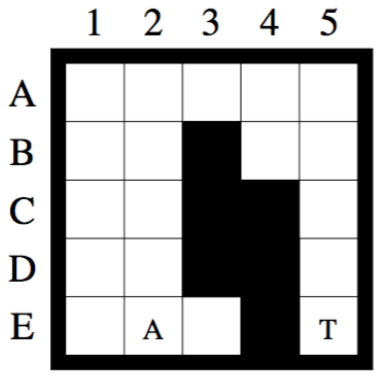


Figure 3: Second Example Search Problem

$h(A_N)$ is calculated as Manhattan distance from A_N to T. Then, we have

$$f(A_N) = g(A_N) + h(A_N) = 4$$

Equivalently, let's denote A_E as the east cell $(E, 3)$ of A, so $g(A_E) = 1$, and $h(A_E) = 2$. Then

$$f(A_E) = g(A_E) + h(A_E) = 3$$

For each iteration in ComputePath, the state with smaller $f-value$ will be chosen from the $OPEN$ list. In this case, A_E is chosen since $f(A_E) < f(A_N)$, the agent will first move to the east rather than to the north.

1.2 Problem B

Problem This project argues that the agent is guaranteed to reach the target if it is not separated from it by blocked cells. Give a convincing argument that the agent in finite gridworlds indeed either reaches the target or discovers that this is impossible in finite time. Prove that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared.

Solution For the first question, a finite gridworld can be seen as an undirected graph, which is divided by the blocked cells into several connected components. The start cell of agent and the goal must be assigned to any of these connected components. In a connected component, every cell is reachable from all other cells in the component. This means an agent can uses search algorithms like DFS or A*

to traverse all the cells in the same connected component in finite time. If start cell and goal cell lie on the same component, the agent can definitely reach the goal, otherwise, it is impossible for agent to reach the goal after all the cells are traversed.

For the second question, suppose there are totally K unblocked cells in a map. Now, let's consider the number of moves in each iteration. For the 1st iteration, the agent will just move from the start cell to a nearby cell, so the number of moves is 1. For the 2nd iteration, the agent may either move to an adjacent new cell or trace back to the start cell and move to the other new cell next to the start cell, so the largest number of moves is 2. Generally, for the ith iteration, the agent will visit exact one new cell, where a "new cell" is said to be the cell that has never been visited in previous $i-1$ iterations. Therefore, the largest number of moves is i. Since there are K unblocked cells, there are only $K-1$ iterations. The worst case is that in each iteration, the agent always traverse the longest path to reach the new cell. Let's denote $N(k)$ as the total number of moves after ith iteration and we have

$$N(k) = \sum_{i=1}^{i=k-1} = \frac{k(k-1)}{2} = O(k^2)$$

If the start cell and the goal cell are in the same connected component of the map, $k = K$ so that $N(k) = N(K) = O(K^2)$, otherwise, $k < K$ so that $N(k) < N(K) = O(K^2)$. Therefore, in the end, the number of moves of the agent is bounded from above by the number of unblocked cells squared.

Part 2 The Effects of Ties

Problem Repeated forward a* needs to break ties to decide which cell to expand next if several cells have the same smallest f-value. It can either break ties in favor of cells with smaller g-values or in favor of cells with larger g-values. implement and compare both versions of repeated forward a* with respect to their runtime or, equivalently, number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation.

Solution After running two algorithms respectively for 50 random maps of size 101×101 (with discrete obstacles), the result is shown in table 1. (Please refer to appendix a for full list of results.)

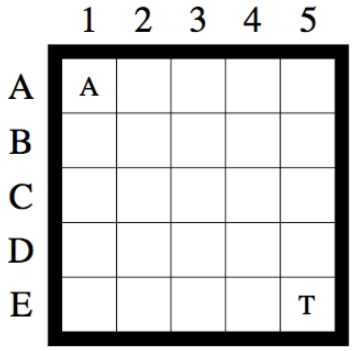
Table 1: Result of two repeated forward a* algorithms

Algorithm	Aver. Expa/M	aver. Expa/P	Aver. Expl/M	Aver. Expl/P
Raw RFA*	264141.78	3058.58	285757.42	3308.32
Modified RFA*	8847.88	92.44	25461.42	267.68
Algorithm	Aver. Moves	Aver. Count	Aver. Optimal	Aver. Ratio
Raw RFA*	335.72	88.2	198.6	1.6852
Modified RFA*	367.56	96.04	198.6	1.8458

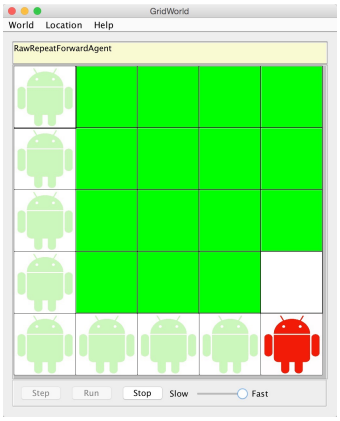
In the table, raw RFA* is the RFA* algorithm that breaks ties by f value and the modified RFA* is the one that breaks ties by $c \times f - g$. the meaning of each column is listed as below:

- Expa/M: total number of cells expanded in each map;
- Expa/P: average number of cells expanded in each plan;
- Expl/M: total number of cells explored in each map;
- Expl/P: average number of cells explored in each plan;
- Moves: total number of actual moves in each map;
- Count: total number of planning in each map;
- Optimal: number of optimal (minimum) moves in each map;
- Ratio: ratio of moves/optimal;

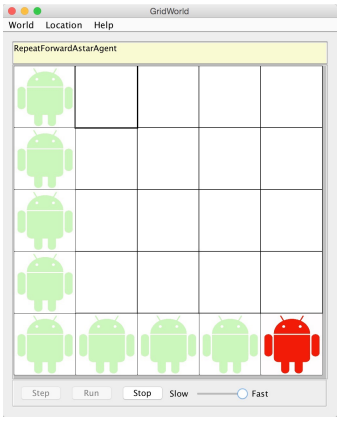
As we can see, the average number of cells expanded in each map by raw RFA* is almost 30 times as that by modified RFA*. Similarly, the average number of cells expanded at each planning (call computepath) by raw RFA* is also 33 times as that by modified RFA*. This indicates that modified RFA* expands much less cells than that by modified RFA*. This is similar to the number of explored cells, which represents all the cells that are added to *open* list and some of them may not be extracted to expand. We can see that the average number of explored in each map by raw RFA* is more than 10 times as that by modified RFA*, and ratio of the number for each planning reaches over 12 times. However, when we inspect the average number of moves, the situation is quite different. Thanks to the large-scale nodes expanded, raw RFA* enable agent to move nearly 9% less steps than modified RFA* either in each map or for each planning. This indiates that the move part of raw RFA* is more likely to approach optimal.



(a) Third Example of Search Problem



(b) RFA* breaking ties by f



(c) RFA* breaking ties by $(c \times f - g)$

Figure 4: Results of two versions of Repeated Forward A*

To explain in detail, let's introduce a simple example as shown in figure 4a. It is a 5×5 grid map where the start cell is at $(A, 1)$ and the goal cell is at $(E, 5)$. We can see that Raw RFA* in Figure 4b almost expands all cells on the map while Modified RFA* in Figure 4c only expands some useful cells. The reason for this difference depends on the way they break ties. In Figure 4b, the algorithm only use f-value to determine the order of expansion which causes a lot of tie cases. For example, at first, the agent will explore its neighbors $(A, 2)$ and $(B, 1)$, whose g-values are both set as 1, h-values are 7 and f-values are 8. The next node to expand is either $(A, 2)$ or $(B, 1)$, so without loss of generality, let's pick $(B, 1)$. After that, cell $(C, 1)$ is explored whose $g = 2$, $h = 6$ and $f = 8$. (Just forget $(B, 2)$ temporarily.) It has the same f-value as $(A, 2)$. In this case, Raw RFA* will pick one arbitrarily, and if it picks $(A, 2)$, $(A, 3)$ will be explored. But for Modified RFA*, it will inspect g value to break ties between $(A, 2)$ and $(C, 1)$ and pick the cell with smaller g, that is $(C, 1)$. Repeatedly, Modified RFA* will expand much less cells than Raw RFA*. Therefore, this is the effect of breaking ties by using $c \times f - g$ value.

Part 3 Forward vs. Backward

Problem Implement and compare Repeated Forward A* and Repeated Backward A* with respect to their runtime or, equivalently, number of expanded cells. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both versions of Repeated A* should break ties among

cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.

Solution

Part 4 Heuristics in the Adaptive A*

Problem The project argues that âĂIJthe Manhattan distances are consistent in gridworlds in which the agent can move only in the four main compass directions.âĂİ Prove that this is indeed the case.

Solution

Problem Furthermore, it is argued that âĂIJThe h-values hnew(s) ... are not only admissible but also consistent.âĂİ Prove that Adaptive A* leaves initially consistent h-values consistent even if action costs can increase.

Solution

Part 5 Heuristics in the Adaptive A*

Problem Implement and compare Repeated Forward A* and Adaptive A* with respect to their runtime. Explain your observations in detail, that is, explain what you observed and give a reason for the observation. Both search algorithms should break ties among cells with the same f-value in favor of cells with larger g-values and remaining ties in an identical way, for example randomly.

Solution

Part 6 Memory Issues

Problem You performed all experiments in gridworlds of size 101×101 but some real-time computer games use maps whose number of cells is up to two

orders of magnitude larger than that. It is then especially important to limit the amount of information that is stored per cell. For example, the tree-pointers can be implemented with only two bits per cell. Suggest additional ways to reduce the memory consumption of your implementations further. Then, calculate the amount of memory that they need to operate on gridworlds of size 1001×1001 and the largest gridworld that they can operate on within a memory limit of 4 MBytes.

Solution The data structure of states in our project show below.

state			
type	name	bits	description
Integer	column	10	the column of state in grid
	row	10	the row of state in grid
Boolean	bSearch	1	reflect whether the cell of this state have been found
Boolean	canMove	1	reflect whether the cell of this state is viable

Denote s as the the space of each state, S as the space of the array of states.Using this data structure, s = 22 bits. In the map of size 1001×1001, then

$$S = s * 1001 * 1001 = 2.6Mbyte.$$

Denote g(s) as the cost from start state to state s, during each iteration, namely the procedure of compute path, the array g need to store all notes which have been explored. Denote h(s) as the estimate cost from state s to goal state, during each iteration, namely the procedure of compute path, the array h need to store all notes which have been explored. Denote search(s) as the explore times of state s, during each iteration,the array s need to store all notes which have been explored. Denote close(s) as the state which have been expanded,during each iteration, the array cost close need to store all states which have been expand. Denote open(s) as the state s will be expand in a priority heap sorted by the sum of heuristic value and g(s). The heap open need to store nearly all notes which have been explored. The type of the value of each elements in the data structure above is integer. Assume the start state and goal state are in the opposite endpoint in the diagonal line of the map. Then each iteration, the number of expand will near the sum of rows and columns, which rows is the total number of rows in map and columns is the total number of columns. So the number of expand ≈ 2000. Then

$\times$ 1001, then

$$g = 4 * 2000 * (32 + 34) \approx 65KByte$$
$$h = 4 * 2000 * (32 + 34) \approx 65KByte$$
$$search = 4 * 2000 * (32 + 34) \approx 65KByte$$
$$close = 2000 * 34 \approx 8KByte$$
$$open = 4 * 2000 * 34 \approx 65KByte$$
$$tree = 4 * 2000 * (32 + 34) \approx 65KByte$$

so, the total space used: $S_{total} = S + g + search + close + open + tree \approx 3MByte$, which is less than 4 Mbyte.Then the project can operate within a memory limit of 4 Mbyte, using a map of size 1001 $\times$ 1001.

References

[1] CollegeBoard. AP central gridworld case study. `http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/151155.html`. [Online; accessed 2-October-2015].

[2] Sven Koenig and Maxim Likhachev. Real-time adaptive a*. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288. ACM, 2006.

[3] Wikipedia. Maze generation algorithm — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Maze_generation_algorithm&oldid=679876968`. [Online; accessed 4-October-2015].

Appendix A Part 2 - Data

The following table shows the result of Repeated Forward A* with f-value to break ties:

Expa/M	Expa/P	Expl/M	Expl/P	Moves	Count	Optimal	Ratio
193939	2731	210778	2968	308	72	198	155%
241247	3133	260651	3385	324	78	202	160%
290740	3303	313775	3565	340	89	208	163%
261493	3076	282159	3319	364	86	198	183%
265125	3535	285301	3804	260	76	200	130%
269247	3496	289484	3759	310	78	198	156%
294643	3202	317934	3455	340	93	198	171%
381789	3601	410470	3872	414	107	206	200%
233863	3118	252433	3365	318	76	196	162%
157708	2160	172349	2360	282	74	198	142%
329602	4336	353612	4652	322	77	198	162%
329680	2536	358172	2755	454	131	198	229%
269377	2960	291730	3205	308	92	198	155%
305818	4023	330092	4343	332	77	196	169%
323755	3205	350503	3470	386	102	196	196%
311526	3943	335212	4243	284	80	198	143%
293787	2937	317747	3177	340	101	198	171%
203746	2546	221013	2762	332	81	196	169%
263649	3337	285326	3611	296	80	196	151%
248162	3141	268052	3393	324	80	198	163%
300202	2943	325008	3186	336	103	196	171%
299406	3402	323624	3677	370	89	196	188%
212412	2441	230413	2648	340	88	198	171%
254674	3690	274313	3975	248	70	198	125%
287253	2762	310868	2989	364	105	198	183%
247032	2839	268384	3084	348	88	202	172%
186408	2662	202645	2894	276	71	196	140%
261635	2400	284418	2609	390	110	200	195%
248697	3552	268251	3832	298	71	196	152%
234020	3250	253314	3518	286	73	196	145%
357994	3729	387464	4036	368	97	196	187%
225317	3086	244718	3352	284	74	202	140%
330790	2876	359311	3124	426	116	198	215%
245950	3616	264951	3896	278	69	196	141%

Expa/M	Expa/P	Expl/M	Expl/P	Moves	Count	Optimal	Ratio
268510	3274	289327	3528	316	83	202	156%
292034	3281	316049	3551	340	90	200	170%
213698	3339	230924	3608	274	65	196	139%
310991	2853	336212	3084	402	110	200	201%
294104	3770	316165	4053	260	79	202	128%
353257	2993	383021	3245	416	119	204	203%
251062	3138	271256	3390	290	81	196	147%
229942	2947	248723	3188	306	79	200	153%
188577	2449	205809	2672	340	78	196	173%
320741	2741	346617	2962	440	118	198	222%
237946	2087	259449	2275	408	115	198	206%
238556	2981	258069	3225	320	81	200	160%
237402	2498	257667	2712	378	96	198	190%
189984	2288	207542	2500	318	84	204	155%
169421	2144	184717	2338	346	80	196	176%
250178	2579	271849	2802	382	98	198	192%

The following table shows the result of Repeated Forward A* with $c \times f - g$ to break ties:

Expa/M	Expa/P	Expl/M	Expl/P	Moves	Count	Optimal	Ratio
7448	87	21738	255	326	86	198	164%
10555	89	30036	254	446	119	202	220%
9513	106	27765	311	358	90	208	172%
7750	94	22063	269	380	83	198	191%
10434	98	29746	280	424	107	200	212%
9880	90	27833	255	422	110	198	213%
6112	88	17808	258	306	70	198	154%
10858	100	31267	289	436	109	206	211%
9791	89	28195	258	444	110	196	226%
5626	82	16487	242	284	69	198	143%
7331	83	21034	239	296	89	198	149%
8462	82	24072	236	414	103	198	209%
10084	92	29369	269	346	110	198	174%
8001	91	23474	269	332	88	196	169%
15108	126	43961	369	410	120	196	209%
6856	92	20034	270	328	75	198	165%
7860	92	23003	270	322	86	198	162%

Expa/M	Expa/P	Expl/M	Expl/P	Moves	Count	Optimal	Ratio
6264	79	17840	225	330	80	196	168%
7386	85	21424	249	352	87	196	179%
7664	97	22480	284	316	80	198	159%
7039	95	20654	279	290	75	196	147%
9170	95	26622	277	376	97	196	191%
7522	67	21360	190	462	113	198	233%
7174	96	21094	285	312	75	198	157%
11622	91	31781	250	504	128	198	254%
10035	98	29101	285	412	103	202	203%
7505	92	21883	270	322	82	196	164%
8132	88	23801	258	322	93	200	161%
8670	88	25175	256	354	99	196	180%
8197	87	24013	255	342	95	196	174%
10383	92	30040	268	436	113	196	222%
9311	99	27209	289	362	95	202	179%
13378	107	36611	292	464	126	198	234%
6546	99	19222	291	290	67	196	147%
8897	93	25370	267	344	96	202	170%
8929	105	26054	306	352	86	200	176%
11307	97	29247	252	464	117	196	236%
8363	85	23831	243	396	99	200	198%
9750	105	28543	310	346	93	202	171%
8332	106	24554	314	310	79	204	151%
8016	86	23377	251	352	94	196	179%
10959	116	32145	341	320	95	200	160%
7234	92	21262	272	290	79	196	147%
8182	81	23190	229	394	102	198	198%
8440	83	23530	232	384	102	198	193%
11833	97	34218	282	396	122	200	198%
9418	85	26692	242	416	111	198	210%
7789	82	22661	241	360	95	204	176%
8799	87	25613	256	368	101	196	187%
8479	86	24589	250	366	99	198	184%