AN EVALUATION OR SEARCH ALGORITHMS USING A JAVASCRIPT IMPLEMENTATION

DEPARTMENT OF MECHANICAL ENGINEERING

Author(s): jhedin

Checked by:

Ref: CC-NN-RR
Filename: Search Evaluations.docx
Date: 29-Jan-2015
Total Pages: 11

Abstract

This report details the implementation, and results of an experiment to compare alternative search functions, for a pathfinding problem over a hexagonal grid of varying size and obstacles.

The solution was implemented in javascript, and randomly generates instances of the problem, then solves them with each search method, Breadth First Search (BFS), Depth First Search (DFS), Iterative Deepening Search (IDS), Greedy Best First Search (GBFS), A*, and Hill Climbing Search (HCS).

Through testing, it's apparent that IDS uses very large amounts of time for all types of problems, though at a space benefit in comparison to BFS. Also, the informed searches and HCS all performed very well compared to the uninformed searches in both time and space complexity.

Further work should be done to visualize the algorithms and their effectiveness interactively, compare the results of alternate grid shapes, and the resulting path costs.

Table of Contents

Table of Figures

1 INTRODUCTION

This report details the evaluation of several pathfinding functions over a hexagonal grid: Breadth First Search (BFS), Depth First Search (DFS), Iterative Deepening Search (IDS), Greedy Best First Search (GBFS), A*, and Hill Climbing Search (HCS). The hexagonal grid was configured to be a triangle in two different sizes(n = 10, n = 21) as shown in Figure 1, with a start location (yellow) and an end location (teal). For alternate instances of the problem, the start and end locations are randomized locations in the grid.

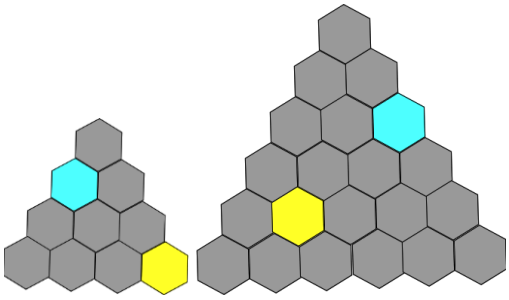

Figure 1 Hexagonal Grids

The given problem is to identify a path from the start location to the end location, while avoiding obstacle edges.

1.1 PROBLEM FORMULATIONS

This problem was formulated as both a search problem and a hill climbing search.

1.1.1 SEARCH FORMULATION

There are several elements defining a search problem:

- the state space,
- the initial state,
- the goal test,
- the successor functions, and
- the solution cost.

For this problem, the state space is the grid locations in the triangle, the initial state is the starting location, the goal test is visited(end location), and the solution cost is the number of grid locations in the path identified from the start location to the end location. Finally, the successor functions are the grid location's neighbors, given as

{(go [i+1, j-1,k], [i+1, j-1,k]), (go [i+1, j,k-1], [i+1, j,k-1]), (go [i-1, j+1,k], [i-1, j+1,k]), (go [i-1, j,k+1], [i-1, j,k+1]), (go [i, j+1,k-1], [i, j+1,k-1]), (go [i, j-1,k+1], [i, j-1,k+1])}, however, depending on blocked edges, not all states can do each of these actions. The state description's cubic coordinates [i,j,k] are described in section 2.2.

1.1.2 HILL CLIMBING SEARCH FORMULATION

Like the search problem, the state of the hill climbing search is the current grid location. The successor function is also the same as the search formulation, to move to another adjacent grid location via an unblocked edge. Finally, the heuristic function is the Manhattan distance to the end location.

1.2 HEURISTICS

Two heuristics ae defined for this problem, the Manhattan distance, and the Euclidean distance. The hexagonal Manhattan distance is described as [abs(Δx) + abs(Δy) + abs(Δz)]/2 in cubic coordinates[1]and the Euclidean distance, as sqrt($\Delta x^2 + \Delta y^2 + \Delta z^2$)

Both these distances are admissible as heuristics as they scale linearly as the state approaches the goal, and don't depend on the path taken to that grid location. Thus, the heuristics never overestimate the actual value.

2 DISCUSSION

2.1 METHODOLOGY

For each of the search methods, 100 problems were solved, under each of 4 configurations,

1) 10 locations, with 20% of the inner edges blocked;

2) 10 locations, with 50% of the inner edges blocked;

3) 21 locations, with 20% of the inner edges blocked; and

4) 21 locations, with 50% of the inner edges blocked.

Since there are a total of 18 inner edges for the 10 locations case, not all of the blocked edge counts ae integers. Thus, for the small grid at 20% blocked edges, the blocked edge count is 3.6. To account for this, 40% of the tests are done using 3 blocked edges, and 60% of the tests are done using 4 blocked edges. Similarly, for the large triangle, there should be 22.5 blocked edges, thus, 50% are done using 22 blocks, and 50% using 23 blocks.

For each test, the following statistics are recorded:

- Total number of visited locations,
- Maximum number of locations stored,
- Runtime(seconds), and
- Number of problems solved

Together, these give approximations of the space and time performance of each algorithm.

2.2 IMPLEMENTATION

The model and search algorithms were implemented in javascript, due to the ease of data storage and testing.

2.2.1 COORDINATE SYSTEM

The states are labeled using a cubic coordinate system as described by Amit Patel at Red Blob Games [http://www.redblobgames.com/grids/hexagons/#coordinates], this uses 3 coordinates, {x,y,z} or {i,j,k} to emulate moving along hexagonal edges as if they were a slice in a 3D grid. This allows for easy adaptation from examples of the search algorithms from square/cubic models into the hexagonal grid.

Valid locations are stored in hash tables, hashing the i,j,k coordinates. In javascript, this is done by defining each coordinate as an object property, which can be accessed using a list [i,j,k] as the property name. This is also done for edges, but using [i1,j1,k1,i2,j2,k2] as the property name for an edge between states 1 and 2.

To generate the triangular grid, axial coordinates were iterated, and then converted to their cubic counterparts to be stored as the grid.

2.2.2 CODE OVEVIEW

2.2.2.1 FILE STRUCTURE

There are currently 3 files used in the structure, index.html to load the javascript files for testing (and for future visualizations), generate.js for creating problems, and seach.js for solving problems.

To create a problem, there ae two steps,

5) Call base = generate(“large”, “triangle”) to create a large triangular grid.

6) Call instance = instantiate(23, base) to create a problem with 23 obstacles.

This was done to allow for future improvements where the searches could be evaluated using different grid shapes, such as a rhombus, or a hexagon, rather than the implemented triangle.

With a problem defined, any of the searches can be performed on it by calling

- BFS(instance) to run BFS,
- DFS(instance) to run DFS,
- IDS(instance) to run IDS,
- GBFS(instance) to run GBFS,
- AS(instance) to run A*, or
- HCS(instance) to run HCS.

Alternatively, the entire test can be run using solve_problems(), and can be printed in tsv format with print_csv().

2.2.2.2 FRONTIER STORAGE

Javascript allows for multiple functions on its arrays, allowing them to act as stacks (push and pop), queues (push and shift), or priority queues (push, sort, pop).

The search code of each method is almost the same, other than the method of storing the frontier:

- BFS uses a queue,
- DFS uses a stack,
- IDS uses a stack with a limited stack size,
- GBFS uses a priority queue,
- A* uses a priority queue, and
- HCS uses a stack where successors for a state are added by priority.

2.2.2.3 SUCCESSORS

Successors are added by calling push_neighbors or push_sorted_neighbors for HCS. This calculates the valid, unvisited successors or the current state, and adds them to the frontier. A tree is also generated by these functions to keep track of the path from start to finish.

To calculate the path, create_path is called, and returns a list of the grid locations from the end to the start location.

2.2.3 INSTRUMENTATION

The search functions return several values,

- The solution path,
- The maximum frontier size, and
- The number of visited states.

The calling function, run_case(), keeps track of the start time, and end time, along with the number of instances solved.

Before testing the algorithms, each problem is tested by BFS to ensure that it is solvable, as the instances are random.

2.3 RESULTS

The results of the experiment are tallied in Table 1.

Table 1 Experimental Results

search	case	problems	Runtime(msec)	space	time
BFS	0	100	16	7.11	5.62
BFS	1	100	15	3.52	4.71
BFS	2	100	78	18.03	11.36
BFS	3	100	66	7.05	10.21
DFS	0	100	16	6.71	6.37
DFS	1	100	15	3.4	4.91
DFS	2	100	47	14.11	11.21
DFS	3	100	31	6.21	9.38
IDS	0	100	110	6.41	18.49
IDS	1	100	47	3.39	13.64
IDS	2	100	293	11.51	40.49
IDS	3	100	187	5.85	35.75
GBFS	0	100	16	5.81	3.12
GBFS	1	100	15	3.45	3.31
GBFS	2	100	16	9.37	4.04
GBFS	3	100	16	5.68	5.29
AS	0	100	15	5.82	3.14
AS	1	100	16	3.46	3.29
AS	2	100	31	9.59	4.38
AS	3	100	31	5.59	5.76
HCS	0	100	16	5.81	3.15
HCS	1	100	16	3.45	3.37
HCS	2	100	15	9.45	4.1
HCS	3	100	31	5.66	5.47

The time and space complexities are plotted in figures 2 – 5; the run time was left out of the comparison, as the IDS results dominated the actual runtime values.

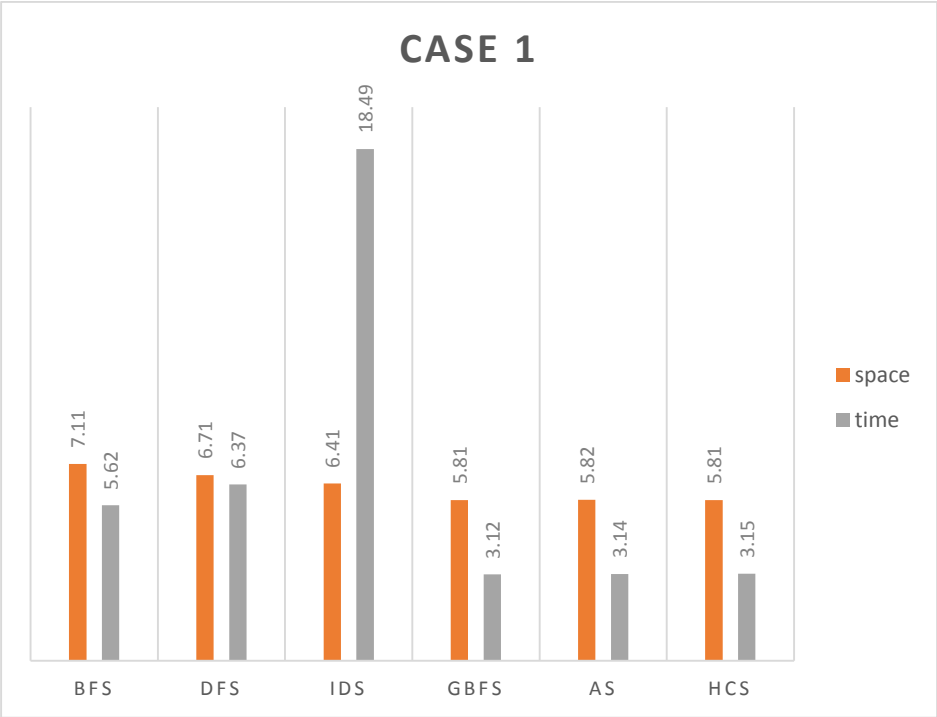

Figure 2 Space and Time Complexity Comparisons for Case 1

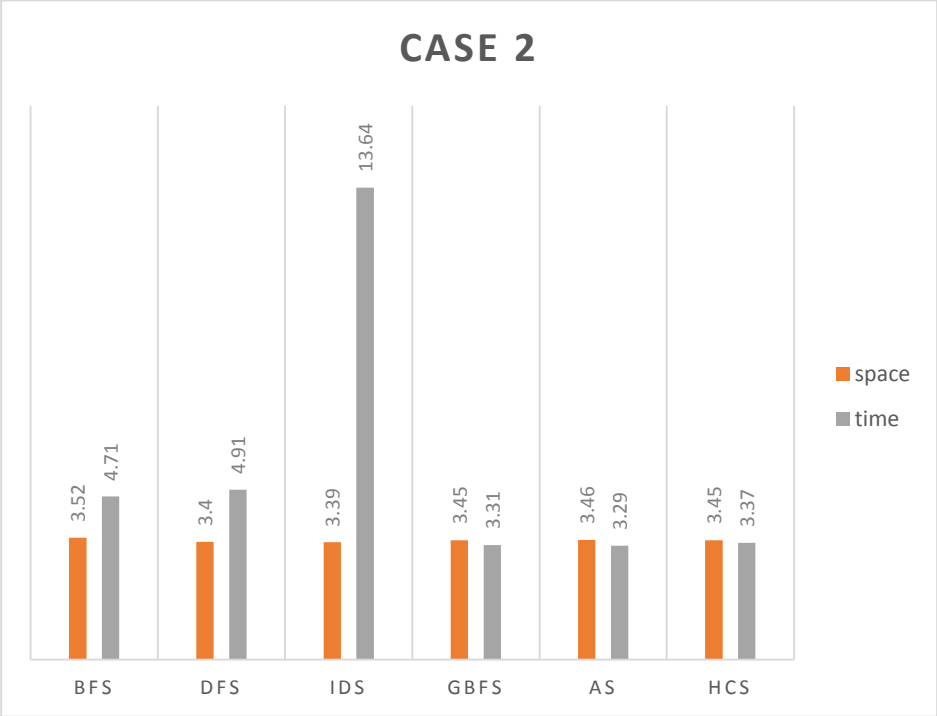

Figure 3 Space and Time Complexity Comparisons for Case 2

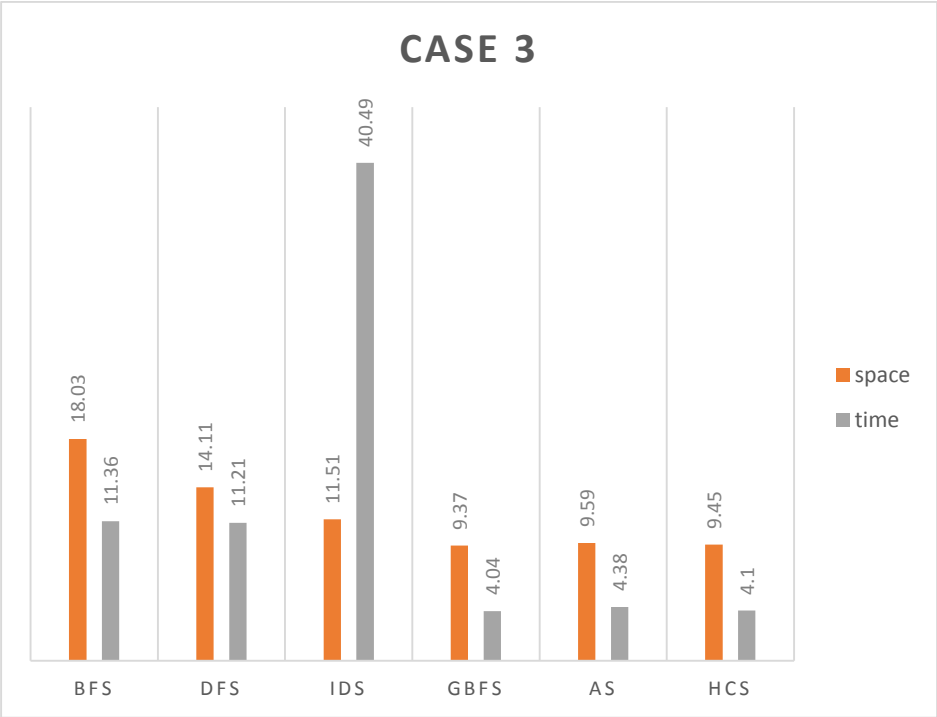

Figure 4 Space and Time Complexity Comparisons for Case 3

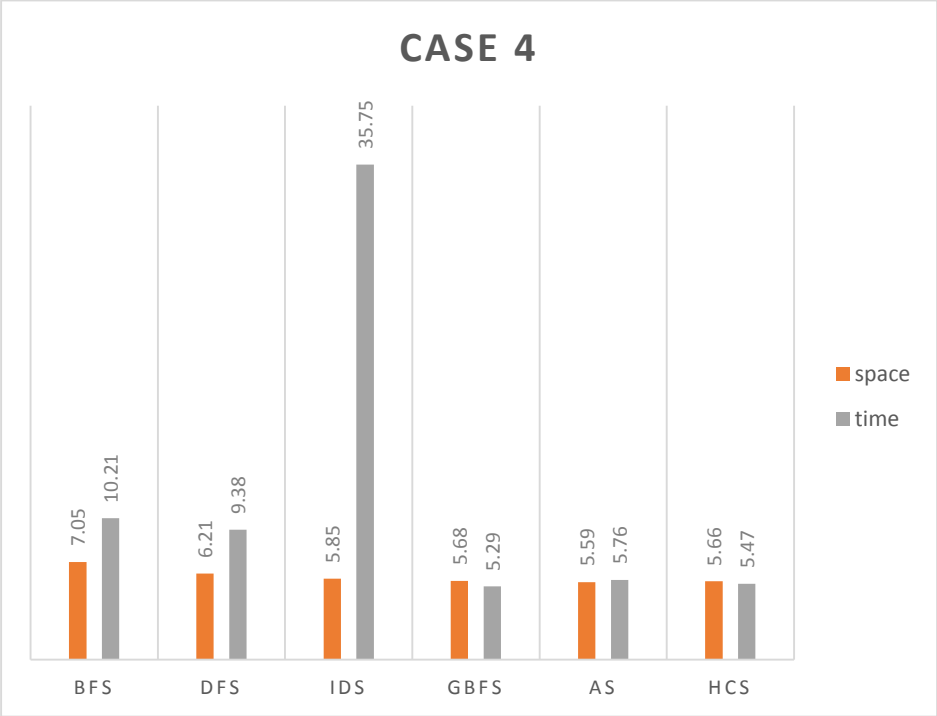

Figure 5 Space and Time Complexity Comparisons for Case 4

From the four figures, some trends are clear; for instance, IDS have very poor time performance, and the informed searches perform very well in time for situations with low obstacles, while also performing better in terms of space complexity than the

uninformed searches. They also do better in cases with high obstacles, though with a much smaller margin. IDS performs as expected for space: it outperforms BFS in all cases.

One surprising result is the similarity of time performance for both A* and GBFS; one would expect A* to outperform GBFS by a small margin when there is high amounts of obstacles. This could be due to generated cases being reasonably well suited for GBFS – roughly straight from the stat location to the end location, as there are only some specific cases of large detours where it performs poorly. Furthermore, there is no comparison of the average path cost, which would likely show the benefits of A* of GBFS.

3 CONCLUSION

Though this experiment, various search algorithms' performances were compared for solving a hexagonal pathfinding problem. The algorithms tested wereBreadth First Search (BFS), Depth First Search (DFS), Iterative Deepening Search (IDS), Greedy Best First Search (GBFS), A*, and Hill Climbing Search (HCS). Of these, IDS performed exceedingly poorly in the number of states visited per solution, while GBFS, A*, and HCS all performed very well in both space and time complexity.

3.1 RECOMMENDATIONS

While this test fits performance expectations, more data should be collected to determine the quality of paths calculated, and to explore the effects of different overall shapes of the grid states. This test assumed a triangular shape, however, the algorithms may perform differently for different boundary conditions, such as a hexagonal or rectangular shape.

Furthermore, it would be useful to implement a visualization of the algorithms using D3.js to plot the exploration/path generation, in addition to plotting the performance characteristics.

4 REFERENCES

[1] *Hexagonal Grids*, Amit Patel, retrieved (2015/01/22)
Link:http://www.redblobgames.com/grids/hexagons/#coordinates

5 APPENDIX

```
// creates a grid of a given shape and [rough] size, with the size
varying to nicely fit the shape
function generate(size, shape)
{
	switch(shape)
	{
		case "triangle":
			return triangle(size);
			break;
		case "rhombus":
			return rhombus(size);
			break;
		case "line":
			return line(size);
			break;
		case "rectangle":
			return rectangle(size);
			break;
		case "hexagon":
			return hexagon(size);
			break;
	}
}

// creates a rectangle shaped grid
function rectangle(size)
{
	small = 12,
	large = 20;

	if(size == "small")
	{
		for(y = 0; y < 3; y++)
		for(x = 0; x < 4; x++)
		{

		}
	}
	else if(size == "large")
	{
		for(y = 0; y < 5; y++)
		for(x = 0; x < 4; x++)
		{

		}
	}
	return {};
}

// creates a triangle shaped grid
function triangle(size)
{
	var locs = {};
```

```
	var node_list = [];
	var small = 10;
	var large = 21;

	locs.imax = 0;
	locs.jmax = 0;
	locs.kmax = 0;

	if(size == "small")
	{
		locs.n = small;
		for(total = 0; total < 4; total++)
		for(split = 0; split <= total; split++)
		{
			// (split, total - split) are the axial coordinates
			var loc = [split, total - split, -total];
			node_list.push(loc);
			locs[loc] = {loc: [split, total - split, -total], cost:
null};
			locs.imax = Math.max(locs.imax,loc[0]);
			locs.jmax = Math.max(locs.jmax,loc[1]);
			locs.kmax = Math.max(locs.kmax,loc[2]);
		}
	}
	else if(size == "large")
	{
		locs.n = large;
		for(total = 0; total < 6; total++)
		for(split = 0; split <= total; split++)
		{
			// (split, total - split) are the axial coordinates
			var loc = [split, total - split, -total];
			node_list.push(loc);
			locs[loc] = {loc: [split, total - split, -total], cost:
null};
			locs.imax = Math.max(locs.imax,loc[0]);
			locs.jmax = Math.max(locs.jmax,loc[1]);
			locs.kmax = Math.max(locs.kmax,loc[2]);
		}
	}
	locs.nodes = node_list;
	return locs;
}

// creates a line shaped grid
function line(size)
{
	small = 10,
	large = 20;

	if(size == "small")
	{
		for(x = 0; x < 4; x++)
		{
```

```
            }
      }
      else if(size == "large")
      {
            for(x = 0; x < 4; x++)
            {

            }
      }
}

// creates a rhombus shaped grid
function rhombus(size)
{
      small = 12,
      large = 20;

      if(size == "small")
      {
            for(y = 0; y < 3; y++)
            for(x = 0; x < 4; x++)
            {

            }
      }
      else if(size == "large")
      {
            for(y = 0; y < 5; y++)
            for(x = 0; x < 4; x++)
            {

            }
      }
}

// creates a hexagonal shaped grid
function hexagon(size)
{
      small = 10,
      large = 19;

      if(size == "small")
      {
            for(y = 0; y < 3; y++)
            for(x = 0; x < 4; x++)
            {

            }
      }
      else if(size == "large")
      {
            for(y = 0; y < 5; y++)
            for(x = 0; x < 4; x++)
```

```
            {

            }
      }
}

// sets the inner obstacles and the begin/end locations
function instantiate(obstacles, base)
{
      var instance = {nodes:JSON.parse(JSON.stringify(base))};
      var edges = {};
      edges.n = 0;

      while(edges.n < obstacles && edges.n < 2*(base.n-1))
      {
            // generate a possible break
            var edge = [base.nodes[Math.floor(Math.random() * base.n)]];
            edge[1] = neighbor(edge[0], Math.floor(Math.random() * 6));
            var swapped_edge = [edge[1],edge[0]];
            // is it valid? is it new?
            if(base[edge[0]] && base[edge[1]])
            {
                  if(edges[edge] || edges[swapped_edge])
                        continue;

                  // add it!
                  edges.n++;
                  edges[edge] = 1;
                  edges[swapped_edge] = 1;
            }
      }

      instance.edges = edges;

      var begin = Math.floor(Math.random() * base.n);
      var end = begin;
      while(end == begin)
            end = Math.floor(Math.random() * base.n);

      instance.begin = base.nodes[begin];
      instance.end = base.nodes[end];

      return instance;
}


function neighbor(current, dir)
{
      var adj =
            [
                  [0,1,-1],[0,-1,1],
                  [1,-1,0],[-1,1,0],
                  [1,0,-1],[-1,0,1]
            ];
```

```
	return [
				current[0] + adj[dir][0],
				current[1] + adj[dir][1],
				current[2] + adj[dir][2]
			];
}

function solve_problems()
{
	base_large = generate("large", "triangle");
	base_small = generate("small", "triangle");
	var case1 = [];
	var case2 = [];
	var case3 = [];
	var case4 = [];

	// instantiate 100 valid problems of each.
	//case 1
	for( var i = 0; i < 40;)
	{
		case1[i] = instantiate(3,base_small);
		if(BFS(case1[i]).path)
			i++;
	}
	for( var i = 40; i < 100;)
	{
		case1[i] = instantiate(4,base_small);
		if(BFS(case1[i]).path)
			i++;
	}
	// case 2
	for( var i = 0; i < 100;)
	{
		case2[i] = instantiate(9,base_small);
		if(BFS(case2[i]).path)
			i++;
	}
	// case 3
	for( var i = 0; i < 100;)
	{
		case3[i] = instantiate(9,base_large);
		if(BFS(case3[i]).path)
			i++;
	}
	// case 4
	for( var i = 0; i < 50;)
	{
		case4[i] = instantiate(22,base_large);
		if(BFS(case4[i]).path)
			i++;
	}
	for( var i = 50; i < 100;)
	{
```

```
            case4[i] = instantiate(23,base_large);
            if(BFS(case4[i]).path)
                  i++;
      }
      var cases = [case1,case2,case3,case4];

      // now, run each set on each case
      var BFS_res = run_cases(BFS, cases);
      var DFS_res = run_cases(DFS, cases);
      var IDS_res = run_cases(IDS, cases);
      var GBFS_res = run_cases(GBFS, cases);
      var AS_res = run_cases(AS, cases);
      var HCS_res = run_cases(HCS, cases);

      return {
            BFS: BFS_res,
            DFS: DFS_res,
            IDS: IDS_res,
            GBFS: GBFS_res,
            AS: AS_res,
            HCS: HCS_res
            };

}

function print_csv(results)
{
      for(var alg in results)
      {
            for(var i = 0; i < 4; i++)
                  console.log(alg +' '+ i +' '+ results[alg][i].problems
+' '+ results[alg][i].sec +' '+ results[alg][i].space +' '+
results[alg][i].time);
      }
}


function run_cases(find_path, cases)
{
      var results = [];
      for(i = 0; i < 4; i++)
      {
            results[i] = run_case(find_path, cases[i]);
      }
      return results;
}

function run_case(find_path, the_case)
{
      var res = {};
      res.space = 0;
      res.time = 0;
      res.sec = 0;
      res.problems = 100;
```

```
	var date = new Date();
	var start = date.getTime();
	for(var inst in the_case)
	{
		var path = find_path(the_case[inst]);
		res.space += path.space;
		res.time += path.visited;
	}
	date = new Date();
	var end = date.getTime();
	res.sec = end - start;
	res.space /= res.problems;
	res.time /= res.problems;

	return res;
}
```

```
function push_neighbors(instance, current, buf, source, visited, cost)
{
	for(var i = 0; i < 6; i++)
	{
		var adj = neighbor(current, i);
		// check that it exists
		if(!instance.nodes[adj])
			continue;
		// check that it's not blocked
		if(instance.edges[[current, adj]])
			continue;
		// this edge isn't new
		if(visited[adj])
			continue;

		source[adj] = current;
		var adj_cost = heuristic(instance, adj) + cost;
		buf.push({loc:adj, cost:adj_cost, integral: cost});
	}
}

function push_sorted_neighbors(instance, current, buf, source, visited,
cost)
{
	var buf2 = [];

	for(var i = 0; i < 6; i++)
	{
		var adj = neighbor(current, i);
		// check that it exists
		if(!instance.nodes[adj])
			continue;
		// check that it's not blocked
		if(instance.edges[[current, adj]])
			continue;
		// this edge isn't new
		if(visited[adj])
			continue;

		source[adj] = current;
		var adj_cost = heuristic(instance, adj) + cost;
		buf2.push({loc:adj, cost:adj_cost, integral: cost});
	}
	buf2.sort(function(a,b){return b.cost-a.cost});
	buf = Array.prototype.push.apply(buf,buf2);
}

function create_path(instance, source)
{
	var path = [];
	var current = instance.end;

	while(source[current] != null)
```

```
	{
		path.push(current);
		current = source[current];
	}
	path.push(current);
	return path;
}

function create_path_from(instance, source, current)
{
	var path = [];

	while(source[current] != null)
	{
		path.push(current);
		current = source[current];
	}
	path.push(current);
	return path;
}

function heuristic(instance, current)
{
	return ( Math.abs(instance.end[0] - current[0]) +
			Math.abs(instance.end[1] - current[1]) +
			Math.abs(instance.end[2] - current[2]) ) / 2;
}

function BFS(instance)
{
	var current = instance.begin;
	var queue = [];
	var source = {};
	var visited = {};
	var max_frontier = 0;
	source[current] = null;
	visited[current] = true;
	push_neighbors(instance, current, queue, source, visited);

	while(queue.length > 0)
	{
		// instrumentation
		max_frontier = Math.max(max_frontier,queue.length);

		// we're done!
		if(visited[instance.end])
		{
			return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
		}
		current = queue.shift().loc;
		visited[current] = true;
		push_neighbors(instance, current, queue, source, visited);
	}
```

```
	// we're done!
	if(visited[instance.end])
	{
		return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
	}
	return false;
}

function DFS(instance, limit)
{
	var current = instance.begin;
	var stack = [];
	var source = {};
	var visited = {};
	var max_frontier = 0;
	source[current] = null;
	visited[current] = true;
	push_neighbors(instance, current, stack, source, visited);
	while(stack.length > 0)
	{
		// instrumentation
		max_frontier = Math.max(max_frontier, stack.length);

		// we're done!
		if(visited[instance.end])
		{
			return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
		}
		if(limit >= 0)
		{
			if(create_path_from(instance, source,current).length >=
limit)
			{
				current = stack.pop().loc;
				continue;
			}
		}
		current = stack.pop().loc;
		visited[current] = true;
		push_neighbors(instance, current, stack, source, visited);
	}
	// we're done!
	if(visited[instance.end])
	{
		return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
	}
	return {path:false, space: max_frontier, visited:
Object.keys(visited).length};
}

function IDS(instance)
```

```
{
	var limit = 0;
	var max_frontier = 0;
	var sum_visited = 0;

	while(limit < instance.nodes.n * 2 - instance.edges.n)
	{
		var path = DFS(instance, limit);
		max_frontier = Math.max(max_frontier, path.space);
		sum_visited += path.visited;
		if(path.path)
			return {path: path.path, space:max_frontier,visited:
sum_visited};
		limit++;
	}
}


function GBFS(instance)
{
	var current = instance.begin;
	var queue = [];
	var source = {};
	var visited = {};
	var max_frontier = 0;
	source[current] = null;
	visited[current] = true;
	push_neighbors(instance, current, queue, source, visited, 0);
	queue.sort(function(a,b){return b.cost - a.cost});
	while(queue.length > 0)
	{
		// instrumentation
		max_frontier = Math.max(max_frontier,queue.length);

		// we're done!
		if(visited[instance.end])
		{
			return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length}
		}
		current = queue.pop().loc;
		visited[current] = true;
		push_neighbors(instance, current, queue, source, visited, 0);
		queue.sort(function(a,b){return b.cost - a.cost});
	}
	// we're done!
	if(visited[instance.end])
	{
		return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
	}
	return false;
}
```

```
function AS(instance)
{
	var current = instance.begin;
	var queue = [];
	var source = {};
	var visited = {};
	var max_frontier = 0;
	var cost_step = 1;
	source[current] = null;
	visited[current] = true;
	push_neighbors(instance, current, queue, source, visited,
cost_step);
	queue.sort(function(a,b){return b.cost-a.cost});
	while(queue.length > 0)
	{
		// instrumentation
		max_frontier = Math.max(max_frontier,queue.length);

		// we're done!
		if(visited[instance.end])
		{
			return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
		}
		var next = queue.pop();
		current = next.loc;
		visited[current] = true;
		push_neighbors(instance, current, queue, source, visited,
next.integral +  cost_step);
		queue.sort(function(a,b){return b.cost - a.cost});
	}
	// we're done!
	if(visited[instance.end])
	{
		return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
	}
	return false;
}

function HCS(instance)
{
	var current = instance.begin;
	var queue = [];
	var source = {};
	var visited = {};
	var max_frontier = 0;
	source[current] = null;
	visited[current] = true;
	push_sorted_neighbors(instance, current, queue, source, visited,
0);
	while(queue.length > 0)
	{
		// instrumentation
```

```
            max_frontier = Math.max(max_frontier,queue.length);
            // we're done!
            if(visited[instance.end])
            {
                  return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
            }
            current = queue.pop().loc;
            visited[current] = true;
            push_sorted_neighbors(instance, current, queue, source,
visited, 0);
      }
      // we're done!
      if(visited[instance.end])
      {
            return {path:create_path(instance, source), space:
max_frontier, visited: Object.keys(visited).length};
      }
      return false;
}
```