CS:4420 Artificial Intelligence
Spring 2017

Course Project

Due: Friday, May 5 by 7pm

1 Project policies

Each team must do a project among those listed, but you are free to choose one to your liking. Every project involves programming and experimental work. *All code must be written in Scala.* More specific instruction on how to submit your source code will be posted on Piazza later.

Each team can decide internally how to divide the project work among its members. However, in addition, each team must officially designate:

- a *coordinator* to organize work sessions, make sure everyone knows where and when to meet and understands who is supposed to be doing what;
- a *recorder* to assemble and turn in the project material;
- a *checker* to check the material to be submitted for correctness and verify that everyone in the group understands both the solutions adopted and the strategies used to obtain them.

Each team member must have one the roles above.[1] Your project report should document who played which role.

All team members are personally responsible for the good functioning of the team and for resolving internal conflicts, especially in the case of non-participating team members. All students will be asked to submit two evaluations (on a form provided by the instructor) of how well they and their teammates performed as team members. Each evaluation is confidential and will not be shown to other team members. The evaluations will be incorporated into the calculation of an individual project grade for each team member.

What to submit. Each team must turn in the following material at the end of the project.

1. A *technical report in PDF format*, describing the project and its results. The report should follow as appropriate A. Sherman and C. Brown's general guidelines for writing tech reports.[2]
2. The *source code of your system*, with detailed documentation and appropriate README files, so as to allow a third party to understand, install and run your code.

[1] In teams of two, one person will take two roles.

[2] Available at `http://www.cs.rochester.edu/u/brown/Crypto/writing/TR_how_to.html` .

3. The set of all *the test files* used to evaluate the system.

All the material must be submitted through ICON as usual but as a zip archive of a single folder called `ProjectMaterial`.

2 Project Selection

2.1 The 8- and the 15-puzzle

Part 1 We saw two heuristic functions for the 8-puzzle: Manhattan distance and misplaced tiles. Several heuristics in the literature purport to do better than either of these. See for example [Nil71, MP89, Kor00]. Test these claims by implementing the Manhattan distance heuristics and at least two more heuristics from the literature, and running your implementation of the A* search strategy on a number of test cases.

Choose a goal configuration at will and keep it fixed for all the test cases. For each $n = 1, \ldots 20$, your test set must contain either all problems that are solvable with exactly n moves, or at least 25 of them chosen randomly among the possible ones. Choose a reasonable time limit for all test cases (no less than 10 minutes per test).

The statistics you collect on the various runs must include the number of expanded nodes, the cost of the solution, the actual running time on a specific machine and the effective branching factor. Report average values for each solution depth.

Part 2 The 15-puzzle is analogous to the 8-puzzle except that it has a 4×4 grid instead of a 3×3 grid and 15 tiles instead of 8. A* search is not effective for solving instances of the 15-puzzle because it requires too much memory. Implement one of the improvements of A* described in the text (IDA*, SMA*) and using a heuristics of your choice run a number of test cases on the 15-puzzle.

Use the experience you have gathered in Part 1 to fine-tune your implementation so that it is as fast as possible on the Linux machines in our lab. Report the most difficult problem (in terms of solution depth) that your implementation can solve in 20 minutes.

In both Part 1 and Part 2, except for the data structures that implement a state, the heuristic function and the procedures that manipulate states, your implementation of the search procedure should be independent from the specific domain.

2.2 Path Planning

Consider the problem of finding the shortest path between two points on a plane that has polygonal obstacles, as shown in Figure 1. This is an idealization of the problem a robot has to solve to navigate its way around a crowded environment.

Suppose the state space consists of a $m \times n$ grid of points in the plane. For simplicity, but without loss of generality if we admit one-vertex polygons, assume that the robot always starts on a polygon vertex in the grid and its goal is to reach some other polygon vertex.

It is not difficult to show that the shortest path from one polygon vertex to any other polygon vertex in the grid must consist of straight-line segments joining a number of polygon vertices. Keeping this in mind, define the necessary functions to implement the search problem, including a successor function that takes a vertex v as input and returns the set of vertices that can be reached

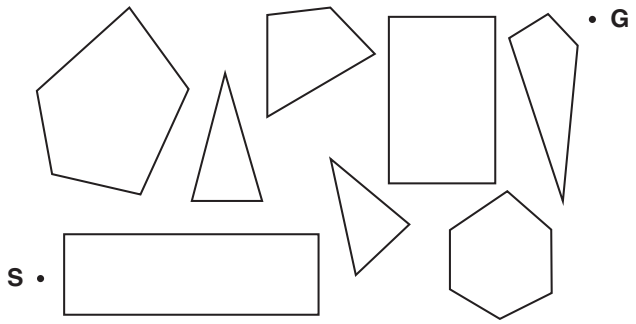


Figure 1: A scene with polygonal obstacles. S and G are respectively the start and the goal state.

from v in a straight line, without intersecting any other vertices or internal points of a polygon in the grid. In this setting a solution is a (possibly non-straight line) between an initial and a final point, described by the sequence of its segments. The cost of a solution is its length.

Part 1 Implement the search algorithm first using the greedy best-first strategy and then any search strategy among A*, IDA* and SMA*. Except for the data structures that implement a state, the heuristic function and the procedures that manipulate states, your implementation should be independent from the specific domain.

Try at least two heuristic functions for your two chosen strategies. You can use any admissible heuristics you deem appropriate for the problem, but you should convincingly argue in your report that the heuristics is in fact admissible.

Use your implementation to run a number of test cases (no less than 20) of different complexity in terms of number, shape and size of the polygons, and difficulty of the solution. Make sure your test cases include non-convex polygons as well.

For each test case report the same kind of statistics required in the 8-puzzle project.

Part 2 Now implement the search algorithm using *k-look-ahead* hill-climbing. This is a variation of hill-climbing that generates the set D of all the descendants of the current node that are at depth $d_c + k$ in the search tree, where d_c is the depth of the current node. Then it chooses as the next node the immediate successor of the current node that is the ancestor of the best node in D.[3] Note that standard hill-climbing is the same as 1-look-ahead hill-climbing.

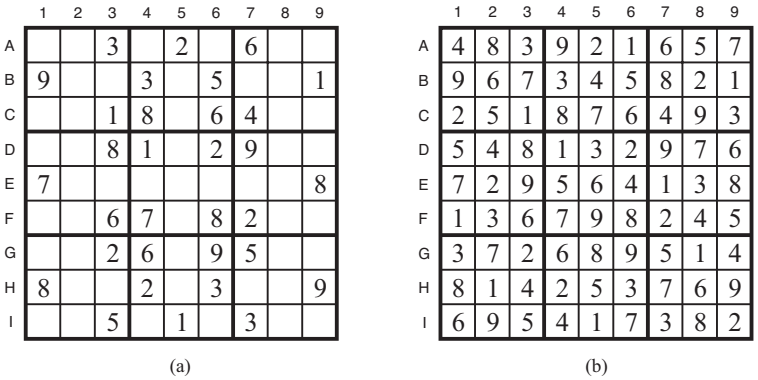
Run this implementation on the same test cases you selected for Part 1 and collect the same sort of statistics for a number of values of the parameter k.

Provide a comparative evaluation of all the various configurations of strategies and heuristics that you tried in Part 1 with the ones in Part 2. Focus on computational and costs and on the quality of the solution found.

Hints

To implement the various domain specific functions in this problem, it is useful to make the following observations.

[3] Provided, of course, that the best node in D is better than the current node.



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 2: A scene with polygonal obstacles. S and G are respectively the start and the goal state.

- A point B is unreachable from a point A if and only if there is a polygon on the grid one of whose sides intersects with an internal point of the segment AB but does not coincide with AB.
- The unique straight line over two points (a_1, a_2) and (b_1, b_2) of the Cartesian plane is denoted by the equation

  $$(b_2 - a_2)x_1 - (b_1 - a_1)x_2 = a_1(b_1 - a_2) - a_2(b_1 - a_1)$$

  in the unkowns x_1 (for the first coordinate) and x_2 (for the second coordinate).
- Two lines $a_{11}x_1 + a_{12}x_2 = b_1$ and $a_{21}x_1 + a_{22}x_2 = b_2$ are parallel iff $a_{22}a_{11} = a_{21}a_{12}$.
- Two lines $a_{11}x_1 + a_{12}x_2 = b_1$ and $a_{21}x_1 + a_{22}x_2 = b_2$ that are not parallel intersect at the point

  $$(a_{22}a_{11} - a_{21}a_{12})^{-1} \cdot (a_{22}b_1 - a_{12}b_2,\ a_{11}b_2 - a_{21}b_1).$$

2.3 The Sodoku Puzzle

Sudoku is extremely popular puzzle with simple rules. The puzzle is a 9×9 integer matrix divided in 9 *regions*, i.e., 3×3 submatrices. Initially, most of the cells in the matrix are empty while the rest contain numbers from the domain $\{1, \ldots, 9\}$. The goal of the puzzle is to fill in the empty cells with integers so that each row, column and region contains every number from 1 to 9.

In general, initial configurations can have zero or more solutions. Sodoku problems, however, consider only initial configurations that have exactly one solution. One example of a Sudoku problem and its solution is given in Figure 2.

Part 1 Model the problem as a constraint satisfaction problem. Then solve each puzzle by using some of the CSP techniques described in Chapter 5 of the textbook. Start with a simple backtracking algorithm. Then improve on it by adding heuristics for variable and value ordering. After that, add constraint propagation techniques such as (some appropriate form of) arc-consistency. If feasible also consider adding conflict-directed backjumping.

You can find a vast collection of Sodoku problems of various difficulty at `http://www.websudoku.com`. Evaluate and compare your various implementations on a number of puzzles selected from each of the four difficulty levels (at least 15 problems per level). Run each puzzle with a timeout of a least 3 minutes. Report statistics similar to those required in the other projects.

Part 2 Search the web for more information on specific techniques for solving the Sodoku puzzle by computer. Implement a selection those and compare your new implementation with the CSP implementations of Part 1, using the same input problems. Analyze and discuss your results.

2.3.1 Hints

- This is a moderately challenging project. To do it well be prepared to spend more time on it that you would on the other projects.
- It you linearize the 9×9 matrix, you can represent the initial configuration of a specific puzzle (or its solution) simply as a string of 81 digits, where 0 can be used for an empty position. This will simplify the input/output interface of your program.
- You may want to write an auxiliary function that checks whether the final configuration found by your program is in fact a solution to the input puzzle. This function is relatively straightforward to code, and will help you debug the main program.

References

[Kor00] Richard E. Korf. Recent progress in the design and analysis of admissible heuristic functions. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 1165–1170, Cambridge, Mass, August 2000. AAAI Press.

[MP89] J. Mostow and A. E. Prieditis. Discovering admissible heuristics by abstracting and optimizing: a transformational approach. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 701–707, San Mateo, CA, August 1989. Morgan-Kaufmann Publishers.

[Nil71] Nils J. Nilsson. *Problem-solving methods in artificial intelligence.* McGraw-Hill, New York, 1971.