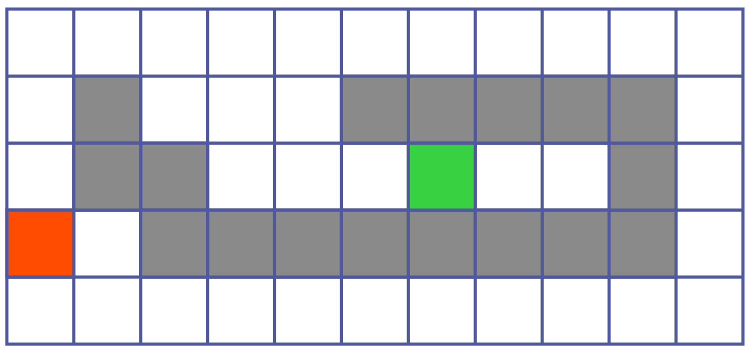
16.410-13 Recitation 9 Problems

Problem 1: A* Search

Robot Navigation *Consider the maze given in the following figure. Name 3 admissible heuristics for this problem. What would be a good admissible heuristics?*



Write down the steps that an A* search would go through using the best heuristic you can think of. You may find it convenient to write the cost-to-go values for each tile on the figure.

Solution One admissible heuristic (that works for every problem) is to write $H(n) = 0$ for all nodes n. This heuristic is clearly admissible, since the actually cost to go is always positive, thus larger than $H(n) = 0$.

A second admissible heuristic is the max distance (also known as the L_∞ distance) of the node to the goal point, computed as follows. Denote the coordinates of the goal by $g = (x_g, y_g)$ and the coordinates of the node n under consideration by $n = (x_n, y_n)$, then the max distance is

$$D(n, g) = \max\{|x_n - x_g|, |y_n - y_g|\},$$

where $|\cdot|$ is the absolute value operator. That is, the L_∞ distance is the maximum travel distance in either direction x or direction y on the map. Clearly, the steps required the get to the goal is at least the maximum of travel in either direction. Thus, this heuristic is admissible.

Finally, a third heuristic is called the Manhattan distance (also known as the taxicab distance or L_1 distance) is computed as follows

$$D(n, g) = |x_n - x_g| + |y_n - y_g|.$$

That is the total travel in either direction. Notice that if there are no obstacles, this heuristic is in fact exact. Clearly, introducing obstacles can only increase the length of the optimal path to get to the goal. Thus, this heuristic is also admissible.

Below, the optimal cost to function is shown (this can be computed by using dynamic programming, e.g., Dijkstra's algorithm, which we will cover later in the course):

8	7	6	5	4	5	6	7	8	9	10
9		5	4	3						11
10			3	2	1	0	1	2		12
11	12									13
12	13	14	15	16	17	18	17	16	15	14

The zero heuristic is given below:

0	0	0	0	0	0	0	0	0	0	0
0		0	0	0						0
0			0	0	0	0	0	0		0
0	0									0
0	0	0	0	0	0	0	0	0	0	0

The L_1 distance heuristic is given below:

6	5	4	5	3	2	2	2	2	3	4
6		4	3	2						4
6			3	2	1	0	1	2		4
6	5									4
6	5	4	3	2	2	2	2	2	3	4

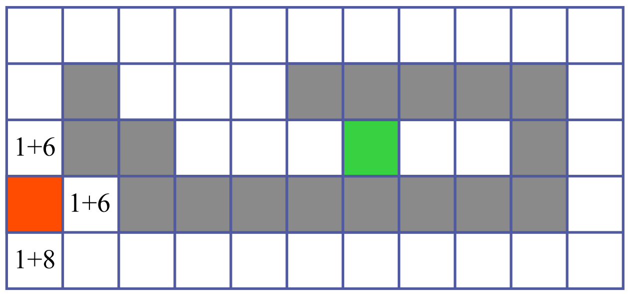
The Manhattan distance heuristic is given below:

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

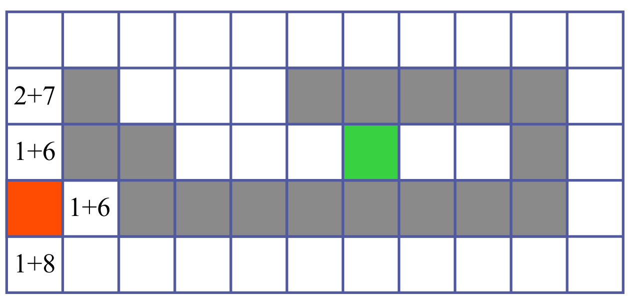
Notice that the values for all the heuristics are always smaller than or equal to the optimal cost-to-go, which confirms that the heuristics are all admissible. Notice also that the Manhattan distance is, in certain places is equal to the optimal cost-to-go. The heuristic based on the L_∞ distance, on the hand, is optimal in much less places, and in many others far away from optimal. Finally, the zero heuristic is never equal to the optimal-cost-go except at the goal.

As a rule of thumb, it is always better to design heuristics that better approximates the cost-to-go. However, the heuristics should also be computationally efficient. Clearly, the zero heuristic is the easiest one to compute. The computational cost for the other two heuristics are almost the same. Hence, we should prefer the Manhattan distance heuristic over the L_1 distance heuristic. We usually evaluate the computational cost of the heuristics using asymptotic analysis, e.g., in this case with respect to the number of nodes in the search graph (i.e., the number of cells). Notice that all the heuristics can be evaluated in constant time with respect to the number of nodes. In other words, no matter how many nodes there are on the network evaluating any of the three heuristics takes some constant number of operations. Thus, we can conclude that these heuristics have essentially the same computational cost (more formal statement would be that the they have the same asymptotic computational complexity with respect to the number of nodes). Clearly, the Manhattan distance heuristic is at least as large as the two others on any node, thus it is more closer to the optimal cost-to-go. Hence, in this problem we prefer to use the Manhattan distance heuristic.

The A* algorithm starts from the initial node shown in red. It evaluates to cost-to-get to each neighboring vertex (in this case each equal to one), then it evaluates the cost-to-go, and it picks the minimum total cost to expand. See the figure below:

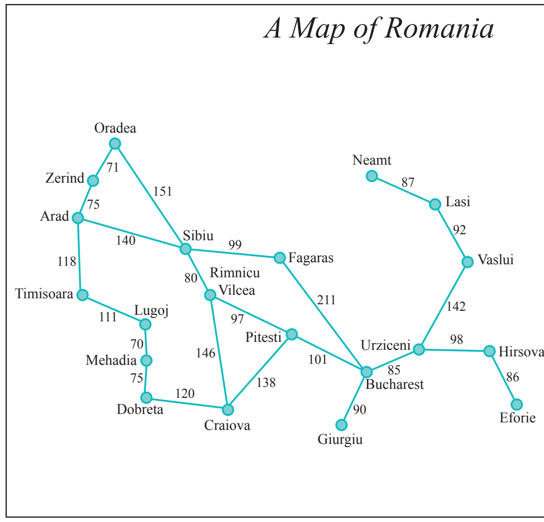


In this case, two nodes have the same total cost, namely 7. The algorithm can proceed with any of these. Let us assume it goes down the top node. There is only one node to expand, which has cost-to-get equal to 2 and cost to go equal to 6. See the figure below



Similarly, the search is continued until all the nodes in the queue are expanded. This is left as an exercise.

Traveling in Romania You are taking your vacation in East Europe. You have just crossed the border from Hungary to Romania. It turned out Romania is a beautiful country.



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Lasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

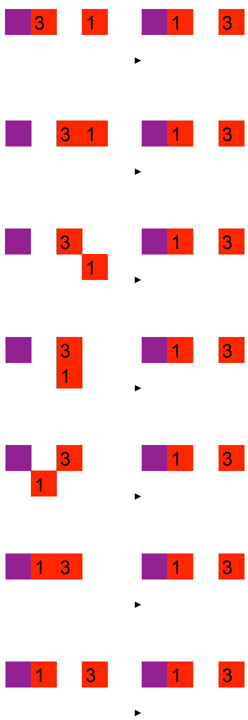
Image by MIT OpenCourseWare.

You have started in Arad on the Hungary border. You would like to see Bucharest. Use the Euclidean distance cost-to-go heuristic in answering the following questions (heuristic values are given above).

- *Find a path that gets you to Bucharest using the Greedy search. Draw the search tree.*
- *Use A* to find an optimal that takes you to Bucharest. Draw the search tree.*
- *Describe how you would construct a "controller" that takes you to Bucharest from anywhere in Romania in case you get lost somewhere. Show the execution of the Dijkstra algorithm.*

Solution This problem is taken from textbook (AIMA, second edition). The solution can be found in the textbook.

Problem 2: Admissible Heuristics



Puzzle Recall the 4-puzzle. Remember the heuristics that you have learned in the class. Describe the Manhattan distance heuristic. Prove that the Manhattan distance heuristic is an admissible heuristic.
Now, consider the following heuristic. Two tiles t_j and t_k are in a *linear conflict* if t_j and t_k are the same line, the goal positions of t_j and t_k are both in that line, t_j is to the right of t_k, and goal position of t_j is to the left of the goal position of t_k. The *linear conflict heuristic* moves any two tiles that are in linear conflict without colliding them. See the figure in left.
Is this heuristic admissible? Either prove that it is admissible, or disprove by a counter-example. Is this heuristic better than the Manhattan distance heuristic.
Rubik puzzle Rubik puzzle is one of the hardest combinatorial problems to date (see the figure below). We would like to solve it using the A* algorithm. What would be a good admissible heuristic for this problem?

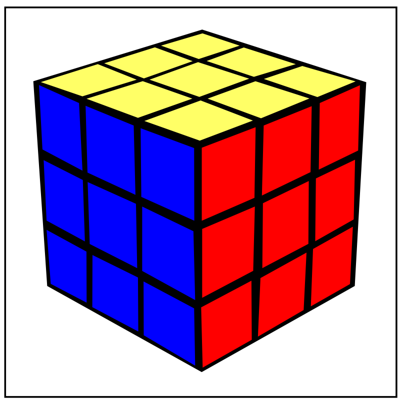


Image by MIT OpenCourseWare.

Solution A configuration of the 4-puzzle is completely described by the the location of each tile, $t_1, t_2, \ldots, t_{15}$. Let us denote the position of a tile t_i by (x_i, y_i) and its goal position by (X_i, Y_i). Then, the Manhattan distance (to the goal) for this particular tile is

$$D_i = |X_i - x_i| + |Y_i - y_i|,$$

and the total distance is

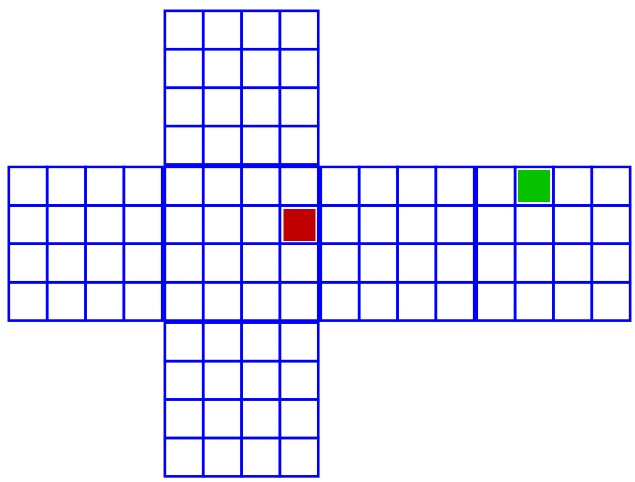
$$D = D_1 + D_2 + \cdots + D_{15}.$$

To reach the goal configuration, tile t_i must travel at least a distance of D_i, which could be achieved if there were no other tiles on the way. Since there are other tiles also, the total distance traveled by the tile t_i while reaching its goal coordinate is at least D_i. Note that this true for each tile, thus the total number of moves for reaching the final configuration from the initial configuration $\{t_1, t_2, \ldots, t_{15}\}$ is at least D. This argument shows that the Manhattan distance heuristic is admissible in this case.

The linear conflict heuristic is indeed admissible. Here we sketch a proof. You are highly encouraged to provide formal proof of this statement. The proof can be constructed based on the simple observation that for any two tiles, say t_i and t_j, the total distance that each of them need to travel to reach their goal positions is no larger than the linear conflict heuristic. In fact, take any two pair of tiles, t_i and t_j. Notice that the linear conflict heuristic adds at most a constant of 2 to the result of the Manhattan heuristic. We examine the two cases separately. First, if the linear conflict heuristic ends up returning the same value that the Manhattan distance heuristic, then the linear conflict heuristic is admissible since Manhattan distance heuristic is also admissible. Second, if the linear conflict heuristic returns a value that is larger than that returned by the Manhattan distance heuristic then the case shown in the figure must have occurred. In this case, since the two tiles can not go over one another, the steps required for both of them reach their goal locations is at least the number returned by the heuristic. Indeed, the actual number of steps is potentially higher since there are more tiles on the way.

Notice that the Manhattan distance heuristic is computed for each tile using information (i.e., the initial position) for that tile only. The linear conflict heuristic indeed considers pairs of tiles. However, computing the Manhattan distance heuristic takes only $O(n)$ time, where n is the number of tiles (in this case $n = 16$). On the other hand, computing the linear conflict heuristic, since it considers all pairs of tiles, takes $O(n^2)$ time. Although, clearly the linear conflict heuristic is more closer to the optimal cost-to-go function, it is computationally more expensive. In this case, we expect the Manhattan heuristic work well in small-sized problems (problems with small n), whereas the linear-conflict heuristic become efficient in larger problems.

The Manhattan heuristic can be extended easily to the Rubik puzzle. Let us represent the tile position by the face that the the tile is on and its four-by-four position on that face, i.e., $t_j = (k_i, x_i, y_i)$, where $k_i \in \{1, 2, \ldots, 6\}$ and $x_i, y_i \in \{1, 2, 3, 4\}$. Given a tile configuration in this form together with the goal configuration (K_i, X_i, Y_i), we consider the length of the shortest path that takes this tile to its final position. One way to compute this distance is to consider unwrap the cube as shown in the figure and compute the number of moves (in the figure, the red tile is the initial position and the green tile is the final position).

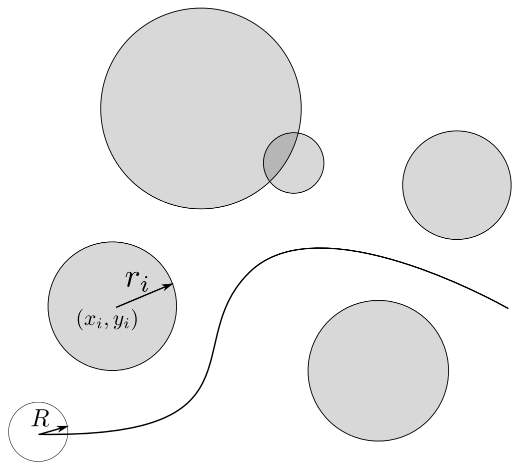


Note that the unwrapping should be done such that the extra face should be on the side that is closer to the initial tile, as seen in the figure.

It is an interesting exercise is to try to extend the linear-conflict heuristic to the Rubik puzzle. Do you think such an extension is possible? If so, can you give an admissible heuristic heuristic based on the linear-conflict heuristic for the Rubik puzzle?

Problem 3: Collision Checking

Recall that sampling-based motion planning algorithms require an "oracle" that checks whether or not a path collides with an obstacle or not. Assume that you have circle-shaped rigid body robot that moves on a plane (2 dimensions). The radius of the robot is R. Each obstacle in the environment is also shaped as a circle. The obstacles are given in the form of a list such that each obstacle is described by the triple (x_i, y_i, r_i), where x_i the x-axis coordinate, y_i is the y-axis coordinate, and r_i is the radius of the obstacle.

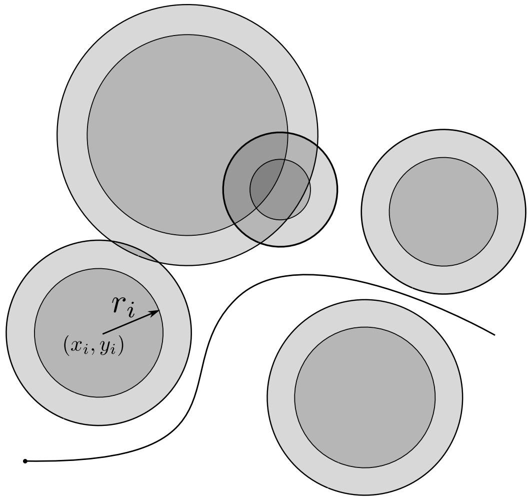


Checking collision with a single obstacle *Devise a method to quickly check whether a given straight path collides with a given obstacle (there is only one obstacle). You can use vector operations (e.g., the dot product or vector multiplications and simple addition and multiplication).*

HINT: Remember the configuration space idea.

Efficiently checking collision with multiple obstacles *Devise a method to store the obstacles in a data structure so that checking collision with obstacles is more efficient than checking collision each obstacle one by one. Analyze the complexity of your collision checking algorithm. How does it scale with the number of obstacles? Compare this to the complexity of checking collision each obstacle one by one.*

Solution To simplify the collision checking process, we exploit the configuration space idea, where the robot is reduced to a point the obstacles are inflated accordingly. Computing the configuration space of a robot may be challenging for complex robot and obstacle geometries. However, in this case, the robot is a disk with radius R and the obstacles are also disks with given radii and position. Clearly, the robot is in collision with an obstacle if its center gets more than R close to that obstacle. Thus the configuration space can be computed by inflating the radius of each obstacle by R. See the figure below. Notice that the robot is reduced to a point (on its center), while the obstacles are inflated by R.



Once the configuration space is computed collision checking can be performed efficiently. Recall that the robot is in collision whenever its center is in collision in the configuration space. The following algorithm checks whether this condition is satisfied or not.

Algorithm 1: `IsRobotInCollision`$\big((x_r, y_r, R),\ (x_1, y_1, r_1), (x_2, y_2, r_2), \ldots, (x_n, y_n, r_n) \big)$

for $i = \{1, 2, \ldots, n\}$ **do**
    $D_i \leftarrow \sqrt{(x_r - x_i)^2 + (y_r - y_i)^2}$ **if** $D_i < r_i + R$ **then**
        return True
return False

The algorithm takes in position and the size of the robot, i.e.,(x_r, y_r, R), and the position and the size of each obstacle, $(x_1, y_1, r_1), (x_2, y_2, r_2), \ldots, (x_n, y_n, r_n)$). It returns true if the robot is in collision, and returns false otherwise.

Notice that for each collision check call, the computational cost of this algorithm is $O(n)$, where n is the number of obstacles. The cost can be reduced considerably, e.g., by using kd-tree type data structure to store the obstacles. The main idea behind the kd-tree is to store each obstacle in a spatial tree structure. These algorithms are known to operate within sublinear time bounds in the worst case. Interested students are referred to the book entitled "The Design and Analysis of Spatial Data Structures," written by H. Samet.

MIT OpenCourseWare
http://ocw.mit.edu

16.410 / 16.413 Principles of Autonomy and Decision Making
Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.