Assignment 1

Heuristic Search

Deadline: Oct. 14 11:59pm
Perfect score: 100 points.

1. Setup

Consider the following problem: an agent operating in a grid-world has to quickly compute a path from the current cell it occupies to a target goal cell. Different cells in the grid have different costs for traversing them. For instance, some may correspond to impassable obstacles, others to flat and easy to traverse regions. There may also be passable but difficult to traverse regions (e.g., rocky, granular terrain or swamps) as well as regions that can accelerate the motion of a character (e.g., highways or navigable regions, etc.).

Similar search challenges arise frequently in real-time computer games, such as Civilization shown in Figure 1(a). Grids with blocked and unblocked cells with different costs are often used to represent terrains in games. Grid-based discretizations are also popular in robotic applications. To control characters in such games, the player can click on known or unknown terrain, and the game characters then move autonomously to the location that the player clicked on. The characters need to quickly compute a path to their final destination and they have many different alternatives in terms of the type of terrain that they can go over. Following a longer path over an easier to traverse terrain may bring the agent faster to the target destination than a short path over a challenging terrain. The A∗ algorithm is mainly used in computing such paths. The original A∗ approach guides the search using an admissible and consistent heuristic. Such a heuristic guarantees that the path returned by A∗ will be optimal.

Figure 1: (a) Games like Civilization utilize an underlying discretization of the environment in order to compute paths (e.g., regular or hexagonal grids), where different cells introduce a different cost for traversing the corresponding terrain, (b) An example of a grid with 8-way-connectivity, where there are obstacle regions (dark), free to traverse regions (white with cost to traverse 1.0), passable but hard to traverse cells (light gray with cost 2.0) and directions where the motion of the agent can be accelerated (blue curve, which decreases the cost of traversing a cell by 4 times).

2. Description of Discretized Maps

We consider a 2D terrain discretized into a grid of square cells that can be blocked, unblocked, partially blocked and directions in the grid where the cost can be decreased as shown in Figure 1(b). White cells are unblocked, light gray cells are partially blocked, dark gray cells are blocked and the blue line indicates a direction of motion along which the motion of the agent can be accelerated (e.g., a "river" in the game of Civilization). Figure 1(b) indicates a shortest path (black line). In the following discussion, the start vertex will be referred to as s_{start} and the goal vertex as s_{goal}.

We will consider grid maps of dimension 160 columns and 120 rows. Initialize these maps by setting all cells in the beginning to correspond to unblocked cells. The cost of transitioning between two regular unblocked cells is 1 if the agent moves horizontally or vertically and sqrt(2) if the agent moves diagonally.

Then, decide the placement of harder to traverse cells. To do so, select eight coordinates randomly (x_{rand}, y_{rand}). For each coordinate pair (x_{rand}, y_{rand}), consider the 31x31 region centered at this coordinate pair. For every cell inside this region, choose with probability 50% to mark it as a hard to traverse cell. The cost of transitioning into such hard to traverse cells is double the cost of moving over regular unblocked cells, i.e.,

- moving horizontally or vertically between two hard to traverse cells has a cost of 2;
- moving diagonally between two hard to traverse cells has a cost of sqrt(8);
- moving horizontally or vertically between a regular unblocked cell and a hard to traverse cell (in either direction) has a cost of 1.5;
- moving diagonally between a regular unblocked cell and a hard to traverse cell (in either direction) has a cost of (sqrt(2)+sqrt(8))/2;

The next step is to select four paths on the map that allow the agent to move faster along them (i.e., the "rivers" or highways). Allow only 4-way connectivity for these paths, i.e., these highways are allowed to propagate only horizontally or vertically. For each one of these paths, start with a random cell at the boundary of the grid world. Then, move in a random horizontal or vertical direction for 20 cells but away from the boundary and mark this sequence of cells as containing a highway. To continue, with 60% probability select to move in the same direction and 20% probability select to move in a perpendicular direction (turn the highway left or turn the highway right). Again mark 20 cells as a highway along the selected direction. If you hit a cell that is already a highway in this process, reject the path and start the process again. Continue marking cells in this manner, until you hit the boundary again. If the length of the path is less than 100 cells when you hit the boundary, then reject the path and start the process again. If you cannot add a highway given the placement of the previous rivers, start the process from the beginning.

In terms of defining costs, if we are starting from a cell that contains a highway and we are moving horizontally or vertically into a cell that also contains a highway, the cost of this motion is four times less than it would be otherwise (i.e., 0.25 if both cells are regular, 0.5 if both cells are hard to traverse and 0.375 if we are moving between a regular unblocked cell and a hard to traverse cell).

Select the set of blocked cells over the entire map. To do so, select randomly 20% of the total number of cells (i.e., 3,840 cells) to mark as blocked. In this process, you should not mark any cell that has a highway as a blocked cell. Agents are not allowed to move into these blocked cells. Paths cannot pass through blocked cells or move between two adjacent blocked cells but can pass along the border of

blocked and unblocked cells. For simplicity, we assume that paths can also pass through vertices where two blocked cells diagonally touch one another.

Finally, place the start vertex s_{start} and the goal vertex s_{goal}. Select the start vertex randomly among unblocked cells (standard or hard to traverse) on the grid in one of the following regions:

- top 20 rows or bottom 20 rows
- left-most 20 columns or right-most 20 columns

Similarly, select the goal vertex randomly among unblocked cells in the above regions. If the distance between the start and the goal is less than 100, then select a new goal.

We assume that the grid is surrounded by blocked cells. We also assume a static environment (i.e., blocked cells remain blocked, unblocked cells remain unblocked, etc.). The agents are occupying and transitioning between the centers of cells. The objective of path planning is to find as fast as possible a short path from s_{start} to s_{goal} in terms of accumulated cost. We will be searching unidirectionally from the start vertex to the goal vertex.

For every map you generate, you should be able to visualize it in a way that the different terrain types are easy to figure out. Furthermore, you should be able to output a file that describes it. You should also be able to load such files into your program. The file should indicate the following information:

- The first line provides the coordinates of sstart
- The second line provides the coordinates of sgoal
- The next eight lines provide the coordinates of the centers of the hard to traverse regions (i.e., the centers (x_{rand},y_{rand}) from the description above)
- Then, provide 120 rows with 160 characters each that indicate the type of terrain for the map as follows:
  - Use ’0’ to indicate a blocked cell
  - Use ’1’ to indicate a regular unblocked cell
  - Use ’2’ to indicate a hard to traverse cell
  - Use ’a’ to indicate a regular unblocked cell with a highway
  - Use ’b’ to indicate a hard to traverse cell with a highway

The following discussion will get into the description of algorithms for computing shortest paths in such grid worlds.

3 Review of A∗

The pseudo-code of A∗ is shown in Algorithm 1 (have in mind that you may be potentially able to improve the speed of your implementation relatively to this pseudo-code description). A* is described in your artificial intelligence textbook and therefore described only briefly in the following, using the following notation:

- S denotes the set of vertices.
- $s_{start} \in S$ denotes the start vertex, and
- $s_{goal} \in S$ denotes the goal vertex.
- c(s, s') is the cost of transitioning between two neighboring vertices s, s' $\in S$ as defined in the previous section.
- Finally, succ(s) $\subseteq S$ is the set of successors of vertex s $\in S$, which are those (at most eight) vertices adjacent to vertex s so that the straight line between s and a vertex in succ(s) is unblocked.

For example, the successors of vertex D3 in Figure 1(b) are vertices C2, C3 and D4. The straightline distance between vertices D3 and C2 is sqrt(2), the straight-line distance between vertices D3 and C3 is 0.25 and the straight-line distance between D3 and D4 is 1. A∗ maintains two values for every vertex s ∈S:

1. First, the g-value g(s) is the distance from the start vertex to vertex s.
2. Second, the parent parent(s), which is used to identify a path from the start vertex to the goal vertex after A∗ terminates.

```
1  Main()
2      g(s_start) := 0;
3      parent(s_start) := s_start;
4      fringe := ∅;
5      fringe.Insert(s_start, g(s_start) + h(s_start));
6      closed := ∅;
7      while fringe ≠ ∅ do
8          s := fringe.Pop();
9          if s = s_goal then
10             return "path found";
11         closed := closed ∪ {s};
12         foreach s' ∈ succ(s) do
13             if s' ∉ closed then
14                 if s' ∉ fringe then
15                     g(s') := ∞;
16                     parent(s') := NULL;
17                 UpdateVertex(s, s');
18     return "no path found";
19 UpdateVertex(s,s')
20     if g(s) + c(s, s') < g(s') then
21         g(s') := g(s) + c(s, s');
22         parent(s') := s;
23         if s' ∈ fringe then
24             fringe.Remove(s');
25         fringe.Insert(s', g(s') + h(s'));
```

Algorithm 1: A*

A ∗also maintains two global data structures:

1. First, the fringe (or open list) is a priority queue that contains the vertices that A∗ considers to expand. A vertex that is or was in the fringe is called generated. The fringe provides the following procedures:
   - Procedure fringe.Insert(s, *x*) inserts vertex s with key *x* into the priority queue fringe.
   - Procedure fringe.Remove(s) removes vertex s from the priority queue fringe.
   - Procedure fringe.Pop() removes a vertex with the smallest key from priority queue fringe and returns it.
2. Second, the closed list is a set that contains the vertices that A∗ has expanded and ensures that A∗ expands every vertex at most once.

A ∗uses a user-provided constant h-value (= heuristic value) h(s) for every vertex s ∈S to focus the search, which is an estimate of the distance to the goal, i.e., an estimate of the distance from vertex s to the goal vertex. A∗ uses the h-value to calculate an f-value f (s) = g(s) + h(s) for every vertex s, which is an estimate of the distance from the start vertex via vertex s to the goal vertex. Upon initialization, A∗ assumes the g-value of every vertex to be infinity and the parent of every vertex to NULL [Lines 15-16]. It sets the g-value of the start vertex to zero and the parent of the start vertex to itself [Lines 2-3]. It sets the fringe and closed lists to the empty lists and then inserts the start vertex into the fringe list with its f-value as its priority [4-6]. A∗ then repeatedly executes the following statements: If the fringe list is empty, then A∗ reports that there is no path [Line 18]. Otherwise, it identifies a vertex s with the smallest f-value in the fringe list [Line 8]. If this vertex is the goal vertex, then A∗ reports that it has found a path from the start vertex to the goal vertex [Line 10]. A∗ then follows the parents from the goal vertex to the start vertex to identify a path from the start vertex to the goal vertex in reverse [not

shown in the pseudo-code]. Otherwise, A∗ removes the vertex from the fringe list [Line 8] and expands it by inserting the vertex into the closed list [Line 11] and then generating each of its unexpanded successors, as follows: A∗ checks whether the g-value of vertex s plus the straight-line distance from vertex s to vertex s 0 is smaller than g-value of vertex s 0 [Line 20]. If so, then it sets the g-value of vertex s 0 to the g-value of vertex s plus the straight-line distance from vertex s to vertex s 0 , sets the parent of vertex s 0 to vertex s and finally inserts vertex s 0 into the fringe list with its f-value as its priority or, if it was there already, changes its priority [Lines 21-25]. It then repeats the procedure. Thus, when A∗ updates the g-value and parent of an unexpanded successor s 0 of vertex s in procedure UpdateVertex, it considers the path from the start vertex to vertex s [= g(s)] and from vertex s to vertex s 0 in a straight line [= c(s, s0)], resulting in distance g(s) + c(s, s0). A∗ updates the g-value and parent of vertex s 0 if the considered path is shorter than the shortest path from the start vertex to vertex s 0 found so far [= g(s 0)]. A∗ with admissible and consistent h-values is guaranteed to find shortest grid paths (optimality criterion). H-values are admissible and consistent (= monotone) if and only if (iff) they satisfy the triangle inequality, that is, iff $h(s_{goal}) = 0$ and $h(s) \leq c(s, s0) + h(s\ 0)$ for all vertices s, s0 ∈S with s 6= sgoal and s 0 ∈succ(s). For example, h-values are consistent if they are all zero, in which case A∗ degrades to uniform-first search.

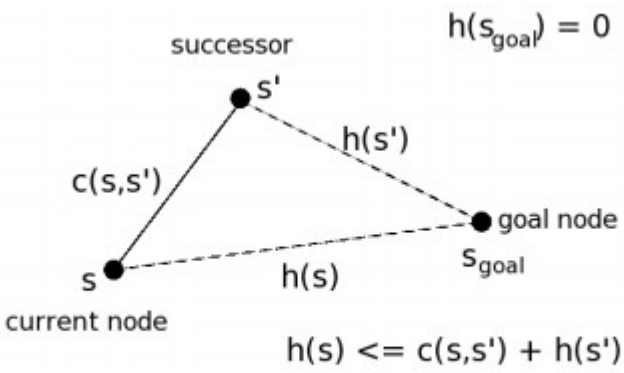

Figure 2: The consistency property for heuristics presented as a triangular inequality.

4 Example Trace of A*

Figure 3 shows a trace of A* over a binary grid. Note that in this example, the vertices correspond to boundary points of cells instead of the center of cells. The algorithms uses as h-values:

$$h(s) = \sqrt{2} \cdot \min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) + \max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) - \min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) \quad (1)$$

s^x and s^y denote the x- and y-coordinates of vertex s, respectively. The labels of the vertices are their f-values (written as the sum of their g-values and h-values) and parents. We assume that all numbers are precisely calculated although we round them to two decimal places in the figure. The arrows point to their parents. Red circles indicate vertices that are being expanded. A* eventually follows the parents from the goal vertex C1 to the start vertex A4 to identify the dashed red path [A4, B3, C2, C1] from the start vertex to the goal vertex in reverse, which is a shortest grid path. *Note that this heuristic may not be admissible or consistent for the problem considered in this programming assignment.*

5 Implementation Aspects

Your implementation of A∗ should use a binary heap to implement the open list. The reason for using a binary heap is that it is often provided as part of standard libraries and, if not, it is easy to implement.

At the same time, it is also reasonably efficient in terms of processor cycles and memory usage. You will get positive experience if you implement the binary heap from scratch, that is, if your implementation does not use existing libraries to implement the binary heap or parts of it.

Do not use code written by others and test your implementations carefully. For example, make sure that the search algorithms indeed find paths from the start vertex to the goal vertex or report that such paths do not exist, make sure that they never expand vertices that they have already been expanded (i.e., follow a GRAPH-SEARCH approach), and make sure that A* with admissible/consistent h-values finds the shortest grid path.

Your implementations should be efficient in terms of processor cycles and memory usage. After all game companies place limitations on the resources that path planning has available. Thus, it is important that you think carefully about your implementations rather than use the given pseudocode blindly since it is not optimized. For example, make sure that your implementations never iterate over all vertices except to initialize them once at the beginning of a search (to be precise: at the beginning of only the first search in case you perform several searches in a row) since your program might be used on large grids. Make sure that your implementation does not determine membership in the closed list by iterating through all vertices in it.

Numerical precision is important since the g-values, h-values and f-values are floating point values. An implementation of A$*$ with the h-values from Equation 1 can achieve high numerical precision by representing these values in the form m + sqrt(2)n for integer values m and n. Your implementations of A$*$ and its variance s can also use 64-bit floating point values (“doubles”) for simplicity, unless stated otherwise.

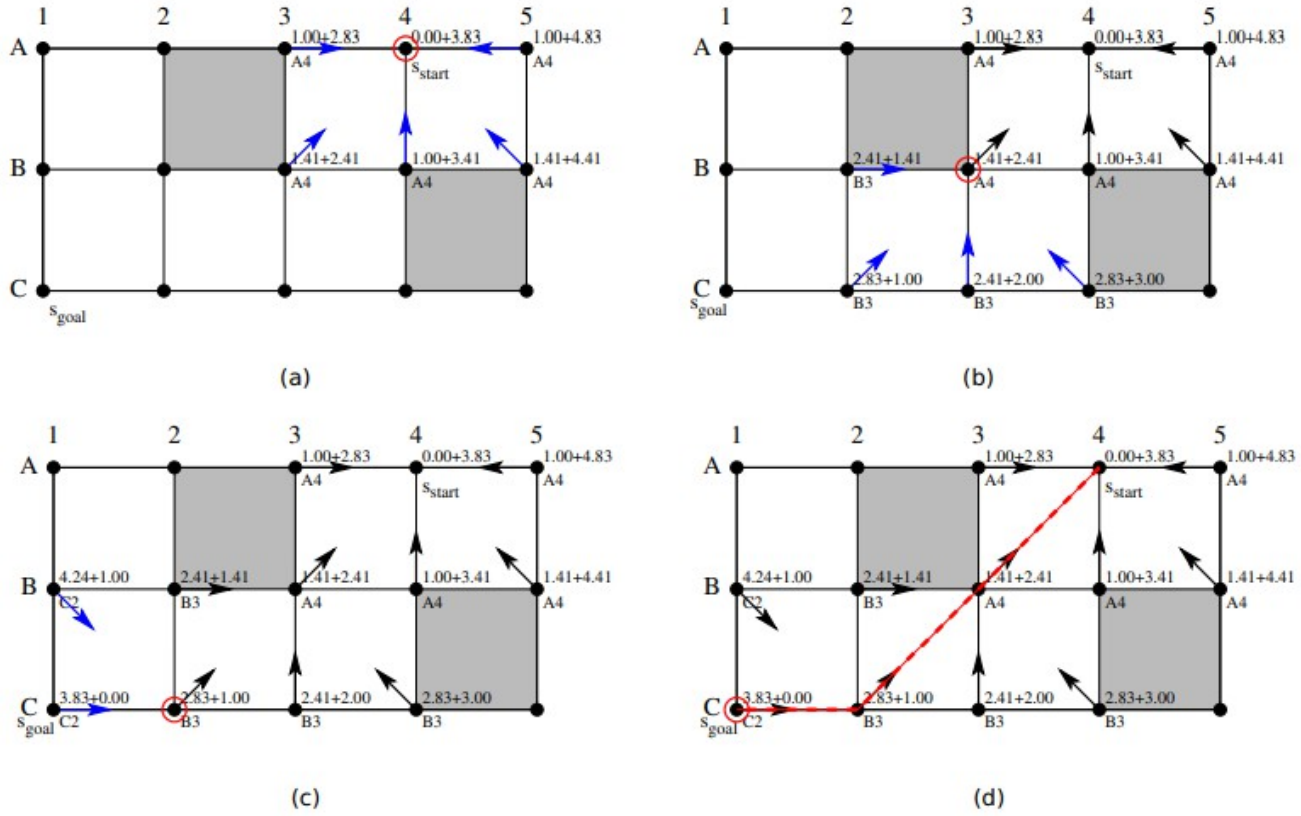

Figure 3: Example Trace of A*

6 Weighted A*

As stated previously, when A∗ is used with a consistent heuristic it is guaranteed that it will return the optimal path from the start to goal state. As with many problems in Computer Science, A∗'s optimality may come with some sacrifices in terms of the cost of computation, a metric for which is the number of expanded vertices (in this way you can compare the performance of different algorithm across different computers). In many cases, one would prefer to trade-off an optimal path with a path that may be suboptimal but which may be calculated faster. This may be achieved if a good heuristic is available and then you can follow a greedier approach that trusts the heuristic more.

In particular, Weighted A∗ expands the states in the order of $f = g + w \cdot h$ values, where $w \geq 1$. In the case that $w = 1$, Weighted A∗ is identical to the original A∗ algorithm. When $w > 1$, the search is biased towards vertices that are closer to the goal according to the heuristic. Weighted A*:

- • trades off optimality for speed when the heuristic is useful; it can actually be orders of magnitude faster than traditional A* given good heuristics, depending on the domain that the search is performed.
- is w-suboptimal, which means that cost(Weighted A∗solution) $\leq w$ ∗cost(optimal solution)

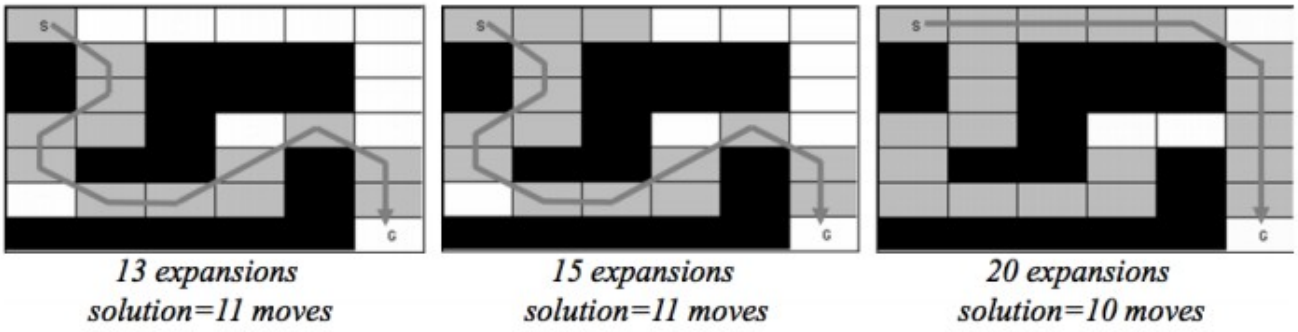

Figure 4: Weighted-A* with: (a) w=2.5, (b) w=1.5, (c) w=1

Figure 4 shows different results from Weighted A∗ depending on the choice of w for the same problem over a binary grid. If a large value for w is chosen then the search is biased towards vertices that are closer to the goal according to a Euclidean distance heuristic (a). If we chose a value closer to 1, then the search is also biased towards the goal but at some point additional vertices are expanded (b). Finally if $w = 1$, we get the optimal path (which is better by 1 move than the suboptimal one) but the number of expansions rises by 7.

Obviously, the effectiveness of A∗ and even more of Weighted A∗ depends on the selection of a good heuristic. Euclidean distance may be a good choice for the example of Figure 4. Nevertheless, the path planning problem described in this project is a little bit more complicated as the underlying grid is not binary. For instance, the Euclidean distance between two nodes (where we assume a distance of 1 between two neighboring horizontal/vertical cells) is no longer an admissible heuristic for our problem (why?). It is rather easy to come up with many types of inadmissible heuristics for this problem, which may still provide some useful guidance towards solving this problem (e.g., Manhattan distance). It is also still possible to come up with an admissible/consistent heuristic for our setup.

A question in this case is which types of heuristics are informative and helpful in finding a good solution fast. Could it be that inadmissible heuristics can still help guide an algorithm to quickly find a solution? You should experimentally figure out and try to verbally describe the trade-offs between using different heuristics and algorithmic choices in the proposed setup.

Next, we will try to investigate a way that allows to integrate multiple heuristics (both admissible and inadmissible ones) into a single search process in a way that potentially improves overall computational performance.

7 A∗ with sequential use of many heuristics

As we mentioned during the lecture, we are typically aiming to discover and use admissible and consistent heuristics during informed search processes so as to achieve optimal solutions. But as you may have also discovered during your experimentation process, there are problems where inadmissible heuristics - which do not guarantee that an optimal solution can be found - can still provide useful information for guiding the search process.

Here we will consider a more sophisticated best-first strategy that has the capability to utilize information from multiple heuristics simultaneously. One of these heuristics has to be admissible and consistent in order to provide some formal guarantees but the rest do not have to be so. Overall, the

algorithm will return a solution with a known error relative to the optimum but will be guided by all heuristics provided to them. You are asked to implement the method, experiment with it and prove some of its properties.

```
1  Key(s,i)
2  |  return g_i(s) + w_1 * h_i(s);
3  ExpandState(s,i)
4  |  Remove s from OPEN_i;
5  |  foreach s' ∈ Succ(s) do
6  |  |  if s' was never generated in the i^th search then
7  |  |  |  g_i(s') = ∞; bp_i(s') = NULL;
8  |  |  if g_i(s') > g_i(s) + c(s, s') then
9  |  |  |  g_i(s') = g_i(s) + c(s, s'); bp_i(s') = s;
10 |  |  |  if s' ∉ CLOSED_i then
11 |  |  |  |  Insert/Update s' in OPEN_i with Key(s', i);
12 Main()
13 |  for i = 0, 1, ..., n do
14 |  |  OPEN_i ← ∅;
15 |  |  CLOSED_i ← ∅;
16 |  |  g_i(s_start) = 0; g_i(s_goal) = ∞;
17 |  |  bp_i(s_start) = bp_i(s_goal) = NULL;
18 |  |  Insert s_start in OPEN_i with Key(s_start, i) as priority
19 |  while OPEN_0.MinKey() < ∞ do
20 |  |  for i = 1, 2, ..., n do
21 |  |  |  if OPEN_i.Minkey() ≤ w_2 * OPEN_0.Minkey() then
22 |  |  |  |  if g_i(s_goal) ≤ OPEN_i.Minkey() then
23 |  |  |  |  |  if g_i(s_goal) < ∞ then
24 |  |  |  |  |  |  Terminate and return path pointed by bp_i(s_goal)
25 |  |  |  |  else
26 |  |  |  |  |  s ← OPEN_i.Top();
27 |  |  |  |  |  ExpandState(s, i);
28 |  |  |  |  |  Insert s in CLOSED_i ;
29 |  |  |  else
30 |  |  |  |  if g_0(s_goal) ≤ OPEN_0.Minkey() then
31 |  |  |  |  |  if g_0(s_goal) < ∞ then
32 |  |  |  |  |  |  Terminate and return path pointed by bp_0(s_goal)
33 |  |  |  |  else
34 |  |  |  |  |  s ← OPEN_0.Top();
35 |  |  |  |  |  ExpandState(s, 0);
36 |  |  |  |  |  Insert s in CLOSED_0
```

Algorithm 2: Sequential Heuristic A*

Consider the method presented in Algorithm 2. This approach aims to utilize information from many different heuristics, which are considered by running n + 1 searches in a rather sequential manner. Each search uses its own, separate priority queue. Therefore, in addition to the different h values, each state uses a different g value for each search. We use g_0 to denote the g for an anchor search process, which must use an admissible/consistent heuristic for the overall process to return an optimal solution. We use g_i, (i = 1, 2, ..., n) for the other search procedures, which can make use of any type of heuristic including inadmissible ones

The algorithm uses two variables in order to control how sub-optimal the overall, final solution will be. The first variable $w_1(\geq 1.0)$ is used to inflate the heuristic values for each of the search procedures, similar to Weighted-A $*$. The second variable $w_2(\geq 1.0)$ is used as a factor to prioritize the inadmissible search processes over the anchor, admissible one. The algorithm runs the inadmissible search procedures in a round robin manner in a way, which guarantees that the solution cost will be within the sub-optimality bound $w_1 w_2$ of the optimal solution cost.

The Sequential Heuristic A$*$ algorithm starts with initializing the variables (lines 13-18) for the different search procedures. It then performs best-first expansions in a round robin fashion from the corresponding queue $OPEN_i$, i = 1..n, as long as $OPEN_i$.Minkey() $\leq w_2 OPEN_0$.Minkey() (line 21). If the check is violated for a given search, it is suspended and a state from $OPEN_0$ is expanded in its place. This in turn can increase $OPEN_0$.Minkey() (lower bound) and thus re-activate the suspended search.

Expansion of a state is done in a similar way as done in A∗. Each state is expanded at most once for each search (line 10) following the fact that Weighted A ∗procedures similar to this one do not need to re-expand states to guarantee the sub-optimality bound. The Sequential Heuristic A∗ terminates successfully, if any of the searches have $OPEN_i$.Minkey() value greater than or equal to the g value of s_{goal} (in that search) and $g(s_{goal}) < \infty$, otherwise it terminates with no solution when $OPEN_0$.Minkey() $\geq \infty$.

8 Questions and Deliverable

In this project you are asked to implement informed search algorithms, to run a series of experiments and report your results and conclusions. You can use the programming language of your preference to implement and visualize the results of your algorithms. You can work in groups of up to 4.

Answer the following questions under the assumption that the algorithms that you will implement are used on an eight-neighbor grids. Average all experimental results over the same set of maps as described in the map generation process and the questions below. You need to generate these maps yourself. Remember that we assumed that paths can pass through vertices where diagonally touching blocked cells meet. All search algorithms search from the start vertex to the goal vertex unidirectionally. Different from our examples, A∗ should breaks ties among vertices with the same f-value in favor of vertices with larger g-values and remaining ties in an identical way. [Hint: Priorities can be single numbers rather than pairs of numbers. For example, you can use $f(s) - c \times g(s)$ as a priority to break ties in favor of vertices with larger g-values, where c is a constant larger than the largest f-value of any generated vertex (i.e. larger than the longest path on the grid).]

You are asked to present the progress of your project to the TAs during an in-person demonstration. Please schedule your meeting as soon as possible once the sign-up link becomes available.

1. Create an interface so as to create and visualize 50 eight-neighbor benchmark grids you are going to use for your experiments, which correspond to:
   - 5 different maps as described above
   - For each map, generate 10 different start-goal pairs for which the problem is solvable.

   Your software should be able to load a file representing a valid map and visualize: the start and the goal location, the different terrain types on the map (e.g., use different colors or symbols) and the path computed by an A∗-family algorithm. You should be able to visualize the values h, g and f computed by A ∗-family algorithms on each cell (e.g., after selecting with the mouse a specific cell, or after using the keyboard to specify which cell's information to display). Use the images in this report from the traces of algorithms as inspiration on how to design your visualization. (10 points)
2. Implement an abstract heuristic algorithm and three instantiations of it: Uniform-cost search, A ∗and Weighted A∗ (it should be easy from the interface to define the weight w). Try to follow a modular implementation so that the code you have to write for each one of the concrete algorithms is minimized relative to the abstract implementation (e.g., take advantage of inheritance in C++ or Java, etc.) Show that you can compute correct solutions with these methods. (15 points)
3. Optimize your implementation of the above algorithms. Discuss your optimizations in your report. (10 points)
4. Propose different types of heuristics for the considered problem and discuss them in your

report. In particular:

- Propose the best admissible/consistent heuristic that you can come up with in this grid world.
- Propose at least four other heuristics, which can be inadmissible but still useful for this problem and justify your choices.

Remember that good heuristics can effectively guide the exploration process of a search algorithm towards finding a good solution fast and they are computationally inexpensive to compute. (10 points)

5. Perform an experimental evaluation on the 50 benchmarks using the three algorithms that you have implemented for the 5 different heuristics that you have considered. For Weighted A ∗you should try at least two w values, e.g., 1.25 and 2 (feel free to experiment). Compare the various solutions in terms of average run-time, average resulting path lengths as a function of the optimum length, average number of nodes expanded and memory requirements (average over all 50 benchmarks). In your report, provide your experimental results. (15 points)
6. Explain your results and discuss in detail your observations regarding the relative performance of the different methods. What impact do you perceive that different heuristic functions have on the behavior of the algorithms and why? What is the relative performance of the different algorithms and why? (10 points)
7. Implement the Sequential Heuristic A∗. Use an admissible heuristic for the anchor search. Use 4 other heuristics, admissible or inadmissible ones, for the remaining search processes. You should try at least two w_1 and two w_2 values, e.g., 1.25 and 2 (feel free to experiment). Make choices so as to optimize the performance of the algorithm, i.e., aim for solutions that compute as high-quality solutions and as fast as possible.

   Perform an experimental evaluation over the 50 benchmarks you have considered above. In particular, compare the A∗ approach with the admissible heuristic against the Sequential Heuristic version, which utilizes all the heuristics you have considered. Compare the various solutions in terms of average runtime, average resulting path lengths as a function of the optimum length, average number of nodes expanded and memory requirements (average over all 50 benchmarks). In your report, provide your experimental results. (15 points)
8. Describe in your report why your implementation is efficient. Explain your results and discuss in detail your observations regarding the relative performance of the methods. Discuss the relationship with your experiments for section e. What is the relative performance of the different algorithm in terms of computation time and solution quality and why? (10 points)
9. In Weighted-A∗ search, the g value of any state s expanded by the algorithm is at most w_1-suboptimal. The same is true for the anchor search of Algorithm 2, i.e., for any state s for which it is true that Key(s, 0) ≤ Key(u, 0) ∀u ∈OPEN0, it holds that $g_0(s) \leq w_1 * c*(s)$, where c ∗(s) is the optimum cost to state s.

   Given this property, show that the anchor key of any state s, when it is the minimum anchor key in an iteration of Algorithm 2, is also bounded by w_1 times the cost of the optimal path to the goal. In other words, show that for any state s for which it is true that Key(s, 0) ≤ Key(u, 0) ∀u ∈$OPEN_0$, it holds that Key(s, 0) ≤ w_1 ∗g ∗(s_{goal}).

   [Hint: Think about a state s_i in the $OPEN_0$ queue, which is located along a least cost path from s_{start} to s_{goal}. Does such a state always exist in the queue? If yes, try to show first that a similar property holds for such a state s_i given the properties of a Weighted-A∗ anchor search. Consider then what this implies for the key of the state s that is the minimum.]

Then, prove that when the Sequential Heuristic A$*$ terminates in the i^{th} search, that $g_i(s_{goal}) \leq w_1 * w_2 * c *(s_{goal})$. In other words, prove that the solution cost obtained by the algorithm is bounded by a $w_1 * w_2$ sub-optimality factor.
[Hint: Consider all cases that the algorithm can terminate] (10 points)