Aesthetic Graph Drawing of Hierarchical State Diagrams

Maximilian Rudolf Hotter, B.Sc.
maximilian.hotter@tum.de

Guided Research
Advisor: Daniel Dyrda, M.Sc.
Supervisor: Prof. Gudrun Klinker, Ph.D.

Chair for Computer Aided Medical Procedures & Augmented Reality
Technical University of Munich
May, 2020

ABSTRACT
This paper discusses an approach to automatically generate edges that represent transitions between fixed, user-created states in a hierarchical state diagram with emphasis on aesthetics and clarity. For this purpose, we applied the graph search algorithm A to a generated grid that uses distance fields as well as different weights and prioritizations for crossing edges and states to calculate different path costs. The results are very promising, and there is a need for further research into post-processing and the avoidance of unwanted edge bending.*

Keywords
aesthetic; visual; graph; graph drawing; hierarchical state diagrams; state diagrams; A* algorithm; A star algorithm; graph search algorithm; Voroni; distance field; edge; node; state; grid; path; path cost; automatic; generation; unmovable nodes

INTRODUCTION

State diagrams are widely used. They are needed to show the behavior of classes in response to events. Depending on what state a system is, it can respond differently to the same event. For example, the numeric keypad on your keyboard can be in two states, depending on whether Num Lock is active or not. If Num Lock is active, the keys 2, 4, 6, and 8 are handled as numbers. If Num Lock is inactive, these keys are handled as arrow keys down, left, right, and up.

Hierarchically state diagrams introduce hierarchically nested states, which means that there are super-states and sub-states. If a system is in a sub-state, it also implicitly is in the corresponding super-state. The system tries to handle all events in the context of the sub-state but if the sub-state is not able to react to a certain event, it passes the event to its super-state, where it is handled in the super-state's context. [1]

To keep an overview of a system with many different states, visualization is very important. It should be legible, and the individual transitions should be clearly identifiable.

Nowadays, the drawing of hierarchical state diagrams is often done manually in order to obtain the most pleasant and, for humans, the most aesthetic and readable graphs. The illustrators decide where to draw the states, where to group them if they wish, and draw the transitions to get visually pleasing results. For this they use aesthetic criteria, which will be presented later in this paper.

Furthermore, there are computer aided graph editors like yEd [2] or Lucidchart [3] that allow users to easily create and manually arrange nodes and edges. The drawback of these solutions is that there are often no fully automated tools that automatically arrange the edges in a pleasing way so that they do not cross or overlap, for example. Nevertheless, they sometimes offer automated assistance like rearranging or manipulating nodes and edges to facilitate this manually.

But there are also graph visualization tools like the open source software Graphviz [4] or PlantUML [5], which take descriptions of graphs in text language together with options like concrete layouts, shapes, colors, styles, fonts, and many more to automatically create diagrams in formats, such as images and SVG. On the one hand, these tools can create visually appealing diagrams that can take up as much space as they need, or the user decided. They can avoid overlapping and even crossing edges. But on the other hand, the disadvantage of this type of software is that users do not have the freedom to take some decisions that could be important to them. An example would be to move the nodes to where they want them to be. Even more significant is the possible transformation of the whole diagram when a new node is added. This can lead to a loss of orientation in the diagram if nodes are moved to other places where they have not been before.

MOTIVATION

All existing solutions found do not offer the capabilities of a state diagram editor, which allows the user to decide where the states are to be drawn and automatically creates smart, fitting edges that represent the transitions between these states with focus on aesthetics and clarity.

Moreover, existing solutions are often designed for planar graphs, and the nodes must be movable by the algorithm. If the formatting of the states is only done by the users, the algorithm should not be able to move states to get promising results. Since finding the perfect location of states is a mental and conscious process of the users, such behavior would probably confuse and annoy them. Consequently, we can no longer guarantee planarity of the diagrams, and crossing or overlapping edges will be unavoidable.

For this reason, we are looking for a new approach that allows users to automatically generate aesthetically pleasing edges for their defined hierarchical states and transitions.

REQUIREMENTS

In this chapter we define the necessary requirements that our approach should meet. They can basically be divided into three main categories: Hierarchical state diagram, grid, and aesthetic criteria.

1. Hierarchical State Diagram

The hierarchical state diagram in our approach consists of *states* and *transitions*.

States are represented as rectangle nodes with variable height and width. Each state can become a *super-state* by adding another nested state to it, which consequently becomes a *sub-state*. Furthermore, multi-level state nesting is possible. A state without any sub-state is also called *atomic state*.

To switch from one state to another, *transitions* are necessary. A transition consists of a *start state*, a *target state*, and a triggering event simply called *trigger* that activates the event and thus the transition from the start state to the target state.

Transitions are represented as edges, which should be automatically generated by our algorithm.

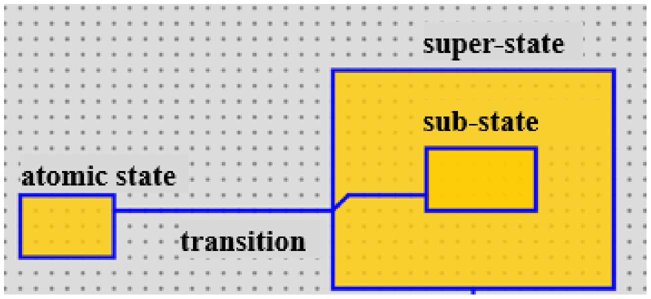


Figure 1. Hierarchical State Diagram with descriptions.

2. Grid

In our approach we use a grid structure, which is made up by straight vertical and horizontal lines. It is used to align the state nodes and transition edges of our hierarchical state diagram. Besides, graph search algorithms can use it for calculating paths from one state to another.

In our example, the grid is visually displayed as dots representing the intersecting points of the grid lines. This grid is located on a canvas on which users can draw their state nodes.

3. Aesthetic Criteria

There are plenty of possibilities how to visualize a graph. Edges are often drawn as curves but can also be straight lines or arrows. Nodes can be either round, square, rectangular, or other shapes. Furthermore, color, thickness, and other graphical properties can be changed.

But even if these characteristics are undeniably important for the legibility of state diagrams, algorithms usually just compute vertex positions of the nodes and curves representing the edges. It is the users' decision how they want to visualize them.

In our example, the state nodes are rectangular, and edges are visualized as several connected horizontal, vertical and diagonal lines.

What is important for us is which specific criteria the algorithm should be able to consider in order to draw the edges.

Battista et al. [6] wrote down some commonly accepted aesthetic criteria for graph drawing:

- Small **number of crossings**,
- Small **drawing area**,

- Small **total edge length**,
- Uniform **edge length**,
- Small number of **edge bends**,
- Large **angular resolution**,
- As **symmetric** as possible.

One of the most important criteria is the small **number of crossings**. Our approach should cause as few crossings and overlaps as possible. They can confuse users and lead to loss of orientation.

In our case, the **drawing area** is only restricted to the canvas size. Nevertheless, we want an edge to not move too far away from the state nodes, for example if they are grouped at one point of the canvas.

The **total edge length** should be kept as small as possible. If the edge lengths become too long, the diagram could appear chaotic and untidy. The same happens if the **edge lengths** are too different and not **uniformed**.

Another important criterion is the small number of **edge bends**. The algorithm should avoid bending the edges when this is not necessary. The edges should not move into dead ends and then move out again.

Since we use horizontal, vertical, and diagonal edges, the **angular resolution**[1] of the graph is not that important. If the edges do not overlap when entering or exiting a state, they can be differentiated well.

The last criterion is to make the graph as **symmetric** as possible. Our approach should not be able to autonomously move state nodes, so this is not possible without the help of the users. But it could handle parallel edges as symmetric as possible.

One criterion that is not mentioned in the list is that we do not want the **edges to be too close to other state nodes** to which they do not belong. On the one hand, it could block possible outgoing edges. On the other hand, it could become confusing if an edge wanders very close around other state nodes. Preferably, edges are supposed to go along the center between two states.

Nonetheless, the computational complexity is mostly NP-hard in general. So, we might never get a perfect result, especially if the computation should be done quickly. Beyond that, it is not possible to simultaneously optimize all aesthetic criteria at once. For example, you cannot always get the smallest total edge length if you want the number of crossings to be zero.

[1] Defined as the sharpest angle between two edges coming from or going to a vertex.

In summary for our approach, we prioritize the following criteria:

1. Edges should not come too close to state nodes to which they do not belong.
2. The number of crossings and overlaps should be as small as possible.
3. Edges should not use unnecessary paths (e.g. into and out of dead ends) and should not move too far from the direct path.
4. Edge lengths should not become unnecessary long.

APPROACH

1. **Graph Search Algorithm**

To find an optimal edge for a transition between two states, we use a graph search algorithm. The edge length and the number of crossings between edges and other edges but also between edges and states should be kept as small as possible to satisfy our aesthetic criteria. Furthermore, the algorithm should be rather performant. It should be able to calculate the edges in a few moments not hours.

a) *Breadth-first Search*

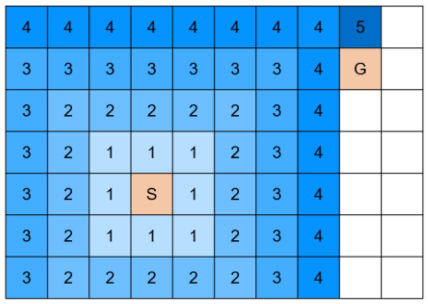


Figure 2. Breadth-first Search starting at node S and finishing early at node G. The depth-level (distance) is displayed in each node.

The Breadth-first search algorithm explores equally in all directions. It starts at a given start point and explores all neighbor points at the present depth level before moving to the next depth level.

The time complexity is $O(|V| + |E|)$, where $|V|$ is the number of vertices (grid nodes) and $|E|$ is the number of edges in the graph. In our approach, each grid node has at least three but a maximum of eight edges [7]. Since we know the number of grid nodes, we could introduce an additional data structure that knows which grid nodes have already been added and explored. This would reduce the space complexity. Furthermore, we can introduce an early exit approach. We do not need to know all the paths to our goal, so we can stop expanding as soon as we have reached the destination. This would result in a time and memory complexity of $O(b^{d+1})$, where d is the distance to the goal from the start and b is the branching factor[2] of the graph [8].

In our case, we cannot use this algorithm properly. Since breadth-first search treats each grid node in the same way, it will always return the same path to its goal, even if we introduce different costs for each grid node. Moreover, this algorithm does ignore the costs at all. Therefore, we cannot use it without further algorithms to prioritize other paths with fewer obstacles (e.g. other paths).

b) Dijkstra's Algorithm

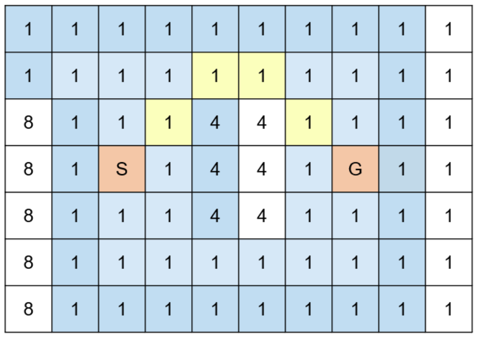


Figure 3. Dijkstra's algorithm finding the shortest path (yellow nodes) on a grid with different movement costs (displayed as number on each node). Blue nodes were explored until the path was found.

The Dijkstra's algorithm, which is also often called Uniform Cost Search, has the benefit of assigning different costs to our grid nodes. Other than the Breadth-first algorithm, it favors lower cost paths and prioritizes grid nodes to explore instead of exploring all nodes equally. We can assign higher costs to grid nodes which contain other paths or represent state nodes, for example. It returns the shortest path, if there are no negative costs.

If there are no negative costs, the optimal runtime is $O(|V|\log(|V|) + |E|)$, where $|V|$ is the number of vertices (grid nodes) and $|E|$ is the number of edges in the graph [9].

In our case this algorithm would give satisfactory results. We can use different movement costs, and it guarantees the shortest and most cost-efficient paths. The disadvantage is that, like the Breadth-first search algorithm, it extends in all directions.

c) Greedy Best-first Search

Since we often have only one path to calculate to our goal, we can introduce a heuristic function that tells us how close we are to the goal. It estimates the costs from one specific node to the goal node (e.g. by using the Manhattan distance).

Instead of ordering the priority queue with the actual distance from the start node, we order the queue by using the estimated distance to the goal. The closest node to the goal is then explored first.

This results in a faster runtime, but the paths are no longer the shortest. Therefore, we cannot use this algorithm if we expect shortest paths and as few bends as possible. In addition, it again ignores the movement costs.

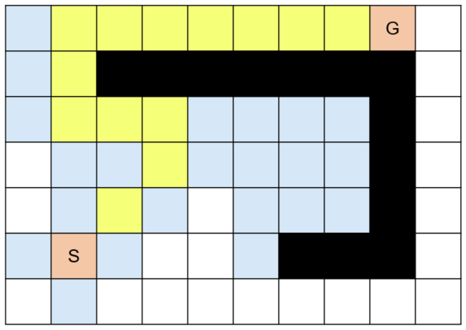


Figure 4. Greedy Best-first Search does not find the shortest path, but it does not need to explore into every direction.

d) A Algorithm*

If we combine the Dijkstra's algorithm and Greedy Best-first Search, we get the A* algorithm. Like Dijkstra's algorithm

[2] Defined as outdegree; the number of edges from each vertex. An average branching factor can also be calculated.

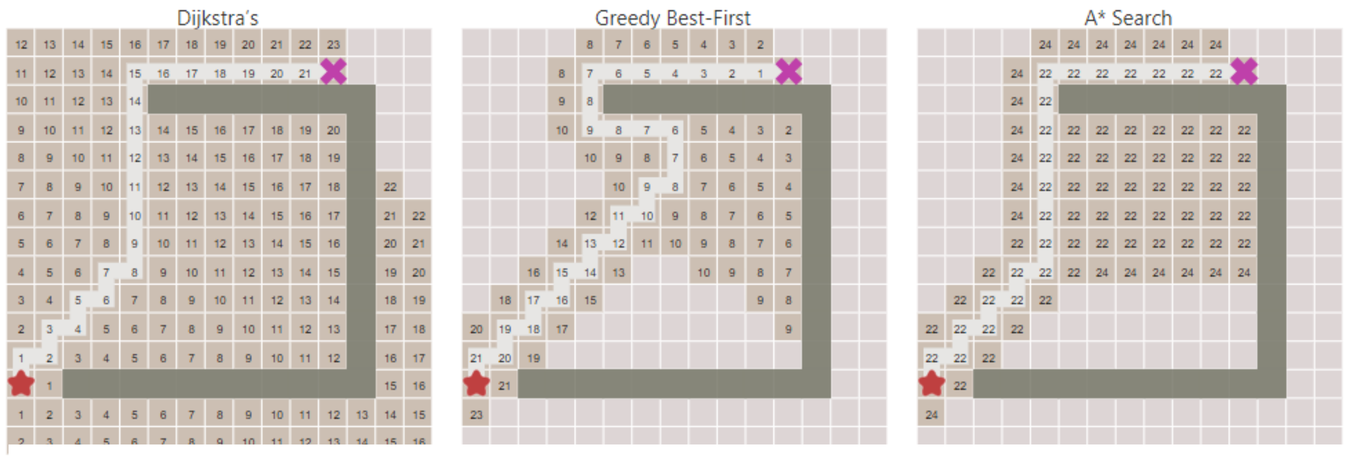


Figure 5. The A* algorithm finds the optimal path like Dijkstra's algorithm, but explores less nodes like Greedy Best-first Search. [12]

it is used to get the shortest path, but it also uses a heuristic to guide itself to the goal.

For this, it uses the sum of the actual distance from the start and the estimated distance to the goal as priority:

$$f(n) = g(n) + h(n)$$

where n is the next node, $f(n)$ is the priority of node n, $g(n)$ is the actual distance (costs) from the start to node n, and $h(n)$ is the heuristic function, thus, the estimated distance (costs) to the goal from node n. The priority is used to order the nodes so that it is more likely to find the goal node. [10]

If the heuristic function does not overestimate the costs, the A* algorithm finds an optimal path. If $h(n) = 0$, the algorithm is equal to the Dijkstra's algorithm. The larger $h(n)$ becomes, the more this algorithm turns into Greedy Best-first Search. If $h(n)$ is larger than the actual distance, the A* algorithm is not guaranteed to find the shortest path. [11]

The time complexity depends on the heuristic function and the implementation of the open and closed lists. The better the expected costs, the less nodes will be explored [13]. Very expensive are operations that include, remove, and change elements in lists. If you use data structures, which can run these operations efficiently the runtime will be shorter. The optimal worst-case time complexity is $O(|V| \cdot \log(|V|))$, where $|V|$ is the number of vertices (grid nodes).

The limiting factor is often memory, since the open and closed list contain every grid node [13]. There are different optimizations and related algorithms that try to solve this problem.

For our approach, we use the A* algorithm as graph search algorithm.

2. Voronoi Tessellation and Distance Fields

According to one of the aesthetic criteria we want the edges to be further away from state nodes they do not belong. It would be desirable for the edges to run centrally between the nodes.

For this purpose, we could think of Voronoi tessellation to calculate the central paths between all state nodes. The vertices of these paths could be then used as anchor points to connect our edges to the center paths of the graph.

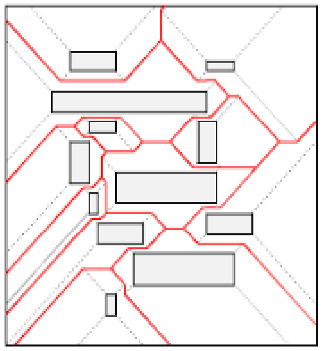


Figure 6. Example Voronoi diagram for rectangles. [14]

To efficiently calculate a Voronoi diagram for rectangular vertices with variable sizes instead of equal sized vertices, Papadopoulou and Lee wrote an algorithm that calculates critical areas for shorts using Voronoi diagrams [14].

Another related approach is using distance fields around our state nodes. If they are big enough, they automatically create a Voronoi field because the movement costs between two nodes is lower the more centered you are.

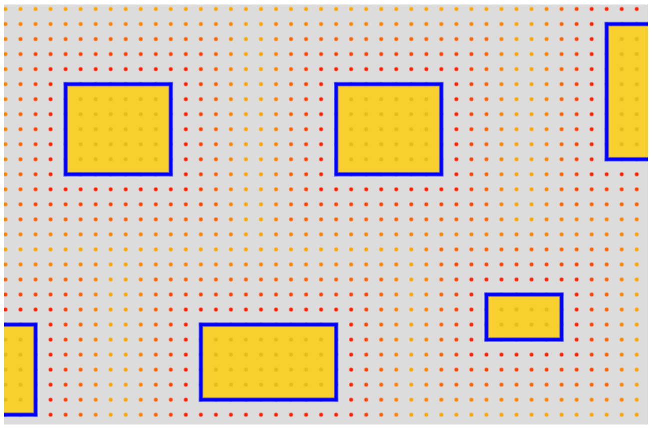


Figure 7. Rectangular vertices with colored distance fields. The brighter the grid points become, the lower the movement costs.

For our approach, we decided to use the distance field because there might be several edges using the same central paths. Thus, you can easily adjust the sizes of the distance fields to get different and hopefully better results.

3. Movement Costs

To meet our aesthetic criteria, we need different movement costs for the A* algorithm to know which grid nodes the algorithm should rather pass.

First, we have different costs for diagonal movement than for horizontal and vertical movement. It should not be as cheap as taking one step horizontally or vertically. Otherwise, the algorithm would prefer diagonal movement. If it is too expensive, the algorithm would rather proceed horizontally or vertically.

A good approach is using the Pythagorean theorem

$$a^2 + b^2 = c^2$$

$$c = \sqrt{a^2 + b^2}$$

where c is the cost of diagonal movement, a is the cost of horizontal movement and b is the cost of vertical movement. If $a, b = 1$, $c = \sqrt{1^2 + 1^2} \approx 1.414$.

There should be also costs for intersecting and overlapping foreign edges, costs for running through foreign states, and staggered costs for moving through foreign distance fields.

4. Prioritization

To avoid intersections and unnecessary long edge lengths, we need a prioritization that decides which edges should be calculated first.

For this, we could either use our heuristic function to calculate the approximate distance or we calculate the real distance by using the A* algorithm without having other edges. While the latter will have much better results for being more precise, it is much more costly. The faster heuristic function, however, is going to ignore other states blocking the direct way, so the actual path could be much longer.

The following applies to all methods: The shorter the edge length, the higher the priority should be.

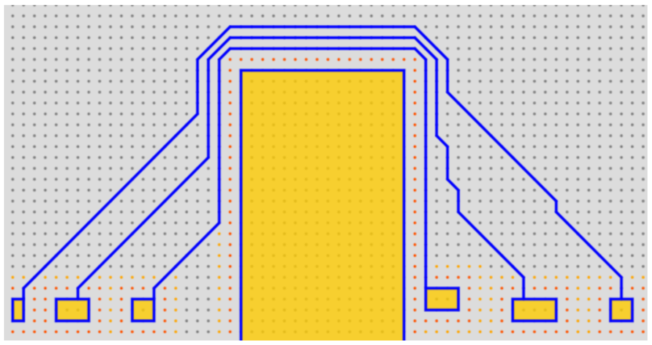


Figure 8. State nodes with shorter edge lengths to each other are prioritized and calculated first. This is the reason why there are no intersections above the large centered state node.

5. Summary

In summary, we have agreed on the following points for our approach:

1. We use the A* algorithm to calculate the optimal paths for our transitions.
2. We use distance fields to move the edges away from other state nodes.
3. We use different movement costs for the diagonal movement than for the vertical or horizontal movement. There are also other costs for intersecting edges, crossing foreign state nodes and distance fields.
4. We prioritize the calculation of edges with shorter length.

IMPLEMENTATION

In this chapter we take a closer look at the implementation of our approach. It was done in JavaScript.

1. Requirements

We have a simple graph editor using a canvas where we can draw rectangular nodes representing our states. Every state has an identifier, a x- and y-position, a width, and a height. They are all stored in a data structure (e.g. Array). It is possible to draw states into other states to represent the hierarchical structure. Furthermore, we are able to define transitions from one state to another. Each transition holds an identifier, a start state, a target state, and the calculated path for the edge. They are also stored in a data structure (e.g. Array).

The canvas is structured as a grid, which is made up by invisible straight vertical and horizontal lines. The intersection points of these lines are called grid points and displayed to the user as dots on the canvas. They are stored as data holders in a two-dimensional data structure (e.g. 2D-Array) and can be referenced by their x- and y-positions. Each grid point knows its x- and y-position, grid point neighbors and the state nodes, distance fields and transition edges lying on it.

The states and later also the edges are aligned to the grid.

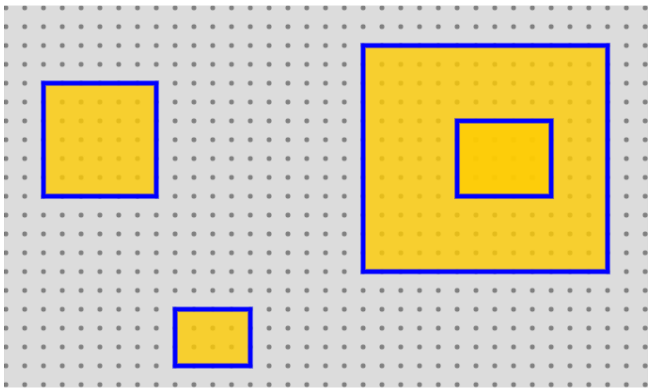


Figure 9. Image of the editor used for this implementation. The states are displayed as yellow rectangles with blue borders. The grid points are shown as grey circular dots.

2. A* Algorithm

There are several ways to implement the A* algorithm. It depends heavily on the available data structures and the information to be stored. The A* algorithm can be very memory intensive, but it can also run slowly if wrong data structures such as simple arrays are used.

The A* algorithm was implemented in the following way:

```
function A_Star(start, goal, heuristic)
  openSet := min-heap containing start
  cameFrom := an empty map

  costSoFar := map (default value of Infinity)
  costsSoFar[start] := 0

  priority := map (default value of Infinity)
  priority[start] := heuristic(start)

  while openSet is not empty
    // This operation needs O(1) time
    current := lowest priority node in openSet
    if current = goal
      return constructPath(cameFrom, current)

    openSet.Remove(current)
    for each neighbor of current
      newCosts := costsSoFar[current] + getCosts(current, neighbor)

      if newCosts < costsSoFar[neighbor]
        cameFrom[neighbor] := current
        costsSoFar[neighbor] := newCosts
        priority[neighbor] := priority[neighbor] + heuristic(neighbor)
        if neighbor not in openSet
          openSet.add(neighbor)
        else
          openSet.update(neighbor)

  return failure
```

Figure 9. Pseudocode of the implemented A* algorithm. [15]

The method *A_Star* expects two grid nodes (*start* and *goal*) and one heuristic function (*heuristic*). *heuristic(n)* estimates the costs from node *n* to *goal*. *A_Star* introduces the following data structures [15]:

- **openSet:** A min-heap containing all discovered nodes that may need to be expanded. At the beginning, it only contains *start*.
- **cameFrom:** A map with nodes. *cameFrom[n]* contains the preceding node on the cheapest path from *start* to *n* currently known.
- **costsSoFar:** A map with all nodes set to *Infinity* except for *start*, which is set to 0. For node *n*, *costsSoFar[n]* is the cost of the cheapest path from *start* to *n* currently known.
- **priority:** $f(n) = g(n) + h(n) = costsSoFar[n] + heuristics(n)$. *priority[n]* contains our current best guess as to how short a path from *start* to *goal* can be if *n* is part of the graph.

While the **openSet** is not empty, the algorithm removes the node with the lowest priority and stores it as *current*.

If *current* is a grid point of our target state, the algorithm has successfully calculated a path from *start* to *goal*.

If not, it calculates the costs (stored in *newCosts)* for every neighbor of *current*. The costs are calculated by adding the costs of going from grid point *current* to grid point *neighbor* and the *costsSoFar[current]*.

If the *newCosts* is lower than *costsSoFar[neighbor]*, the path to the neighbor is better than the previous ones. We store everything and see, whether the *neighbor* has been in the **openSet** before. If not, we add the *neighbor* to the **openSet**. Otherwise, we update the *neighbor's* **priorty**.

If the **openSet** becomes empty, no path from *start* to *goal* could be found.

3. Heuristic Function

To calculate the heuristic, we basically multiply the distance in steps by the minimum cost for a step.

In our approach, we cannot use the Manhattan distance [16] defined as $f(n_i, n_j) = D \cdot (\left|x_{n_i} - x_{n_j}\right| + \left|y_{n_i} - y_{n_j}\right|)$, where $x_{n_i}$is the x-position and $y_{n_i}$is the y-position of node n_i and D is the minimum cost for moving from node n_i to an adjacent node n_j, because it is only used for square grids that allow four directions of movement (up, right, down, and left).

Since our square grid allows eight directions of movement (up, up-right, right, down-right, down, down-left, left, and up-left), we use the Diagonal distance as heuristics function.

To calculate the diagonal distance, we need the following method:

```
function diagonalDistance(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) + (D2 - 2 * D) * min(
dx, dy)
```

Figure 10. Pseudocode for calculating the diagonal distance from *node* to *goal*.

It returns the number of steps we take, if diagonal movement is not allowed, and subtracts the steps we save by using diagonal movement. *D* is the minimum cost for vertical and horizontal movement. *D2* is the minimum cost for diagonal movement. [17]

4. Distance Fields

The distance field with variable size *s* is calculated by moving *s* times around the entire state node and moving one step up as soon as it reaches the start grid point of the last round again. The start grid point of the first round is the upper left grid point of the upper left corner of the state node.

Each reached distance field grid point stores |*r* – *s*| as individual cost, where *s* is the distance field size and *r* the current round starting at 0. It is used to calculate the correct costs for movement on the distance fields.

5. Costs

We implemented modifiable costs for different scenarios that can occur when calculating the edges.

First, we have the cost for **vertical/horizontal** and the cost for **diagonal movement**.

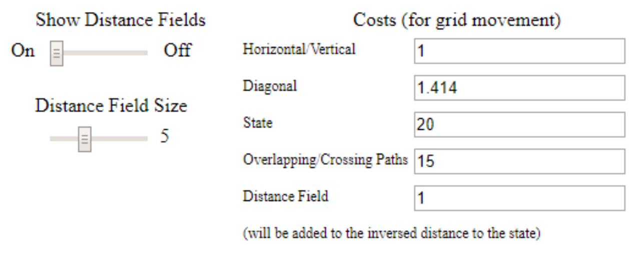
Second, we have the **cost for crossing and overlapping** other edges. This cost gets multiplied by the number of different edges a grid point shares.

Third, we implemented the **cost for movement on states** to which the edge does not belong.

Fourth, we have the **additional cost for movement on distance fields**, which gets added to the **individual distance field grid point cost** mentioned in the Distance Field section of this chapter before. The distance field of the start state is ignored.

The final cost for moving from one grid node to another is calculated by multiplying the cost for diagonal or vertical/horizontal movement by the sum of the various costs mentioned above and 1.

The modifiable costs can be adjusted during runtime below the graph editor.



Show Distance Fields

On Off

Distance Field Size

5

Costs (for grid movement)

Horizontal/Vertical	1
Diagonal	1.414
State	20
Overlapping/Crossing Paths	15
Distance Field	1

(will be added to the inversed distance to the state)

Figure 11. Modifiable costs below the graph editor.

6. Prioritization

Prioritization is done by running the A* algorithm for each transition to sort the transitions by actual cost. Edges of other transitions are ignored and intersecting costs nothing.

After sorting the transitions from low to high cost, the edge drawing process is started in this order and the A* algorithm is run through again for each transition.

RESULTS AND OUTLOOK

The results are very promising. Our approach is not only able to connect all states in hierarchical state diagrams, but also follows our four defined aesthetic criteria with the implementation.

Because of our distance field, **edges do not come too close to state nodes** to which they do not belong. This can be altered by changing the distance field's size or by modifying the additional distance field cost.

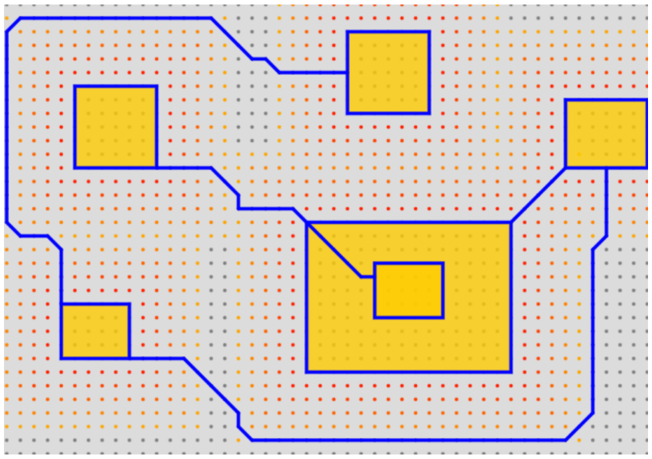


Figure 12. Edges keep distance to nodes.

The **number of crossing and overlaps** is kept as small as possible, as this involves high costs. Of course, this cannot be completely prevented, as this would require dislocation of nodes, but the cost can be adjusted as desired to prevent all unnecessary intersections, whether it is states or edges.

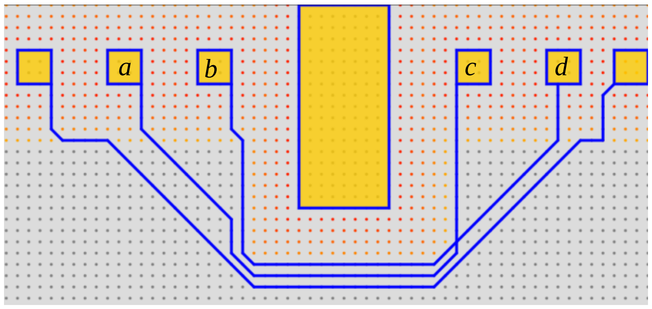


Figure 13. Since there is no way above the centered blocking state, node *a* must cross the path from *b* to *c* to come to *d*. Crossing states has high cost.

By using the A* algorithm, we can guarantee that it will find a shortest path if the heuristic function outputs equal or lower numbers than the actual cost of moving from one grid point to another.

Hence, **edges do not use unnecessary paths** (e.g. into and out of dead ends) and do not **move too far from the direct path**.

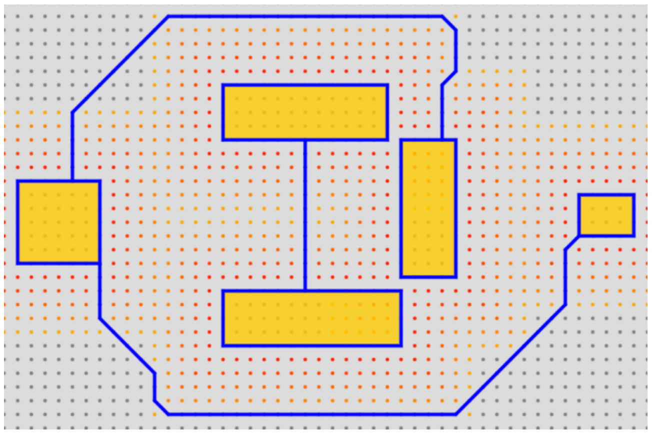


Figure 14. Edges do not lead in and out of dead ends.

Furthermore, by using the A* algorithm, **edge lengths should not become unnecessary long**.

Problems can arise, when making the grid very small. The more grid points are used, the more performance and memory is needed to compute the edges. Our algorithms are not really optimized for that case.

Furthermore, we could think of other grid representations. The A* algorithm runs way faster with fewer possible graph points. Probably using Voronoi for edge "highways" where multiple edges can be combined to a thicker strand could be beneficial for computational complexity.

Another problem is the bends that this approach creates in probably unwanted places (e.g. Figure 12 shows bends in the center of the image that a user would probably not have drawn manually). Squared grids encourage this behavior. There is not much that users can do to currently prevent this. They can try to change the different costs and the distance field size to get a better result for their graphs.

For this purpose, post-processing algorithms (like "string pulling" algorithms or Lerp) could be used to straighten unwanted bends. Active Contour Models ("Snakes") [18] to find smother contours can also be promising.

A different approach is to use this result as a template, extract important points from the edges and use these points for more suitable algorithms for drawing edges. There, possible curved paths with Bezier or similar would be possible.

CONCLUSION

The approach presented in this paper delivers useful results. It respects all the rules and aesthetic criteria we have established and quickly computes the edges during runtime. Further research needs to be done to avoid unwanted bends. Various post-processing methods could be explored to obtain even better edges. Even curved edges using Bezier would be possible.

In terms of performance our approach delivers good results. Nevertheless, further optimizations should be done for bigger grids or other grid representations should be used.

REFERENCES

1. M. Yannakakis, Hierarchical State Machines, in Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, Lecture Notes on Computer Science 1872: Springer-Verlag, 2000, p. 15.
2. Website of the graph editor yEd. https://www.yworks.com/products/yed (last opened April 28, 2020)
3. Website of the graph editor Lucidchart. https://www.lucidchart.com/pages/ (last opened April 28, 2020)
4. Website of the open source software Graphviz. https://www.graphviz.org/ (last opened April 28, 2020)
5. Website of the open source software PlantUML. https://plantuml.com/ (last opened April 28, 2020)
6. Giuseppe di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall, 1999.
7. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. 22.2 Breadth-first search. Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill, 2001, pp. 531–539.
8. Russell, Stuart; Norvig, Peter. Artificial Intelligence: A Modern Approach (2nd ed.). Prentice Hall, 2003, p. 81.
9. Cormen, Thomas H. Introduction to Algorithms. MIT Press, 1990, p. 663.
10. Nilsson, Nils J. The Quest for Artificial Intelligence. Cambridge: Cambridge University Press, 2009.
11. De Smith, Michael John; Goodchild, Michael F.; Longley, Paul. Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools, Troubadour Publishing Ltd, 2007, p. 344
12. Patel, Amit. Introduction to the A* Algorithm: The A* algorithm. Picture was captured from Web Resource. https://www.redblobgames.com/pathfinding/a-star/introduction.html (last opened May 1, 2020)
13. Russell, Stuart; Norvig, Peter. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall, 2009, p. 103.
14. Papadopoulou, Evanthia; Lee, D. Critical Area Computation -- A New Approach. 1999.
15. Adjusted version of the algorithm provided on Wikipedia. A* search algorithm. Description, Pseudocode. https://en.wikipedia.org/wiki/A*_search_algorithm (last opened May 1, 2020)
16. Black, Paul E. "Manhattan distance". Dictionary of Algorithms and Data Structures. https://xlinux.nist.gov/dads/HTML/manhattanDistance.html (last opened May 1, 2020)
17. Patel, Amit. A*'s Use of the Heuristic. https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html (last opened May 1, 2020)
18. Kass, Michael; Witkin, Andrew; Terzopoulos, Demetri. Snakes: Active Contour Models. International Journal of Computer Vision, 1988, pp. 321-331.