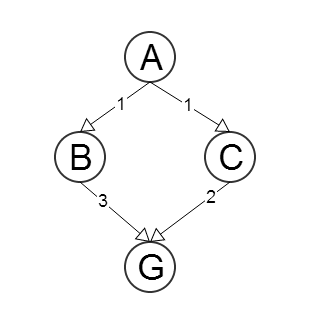
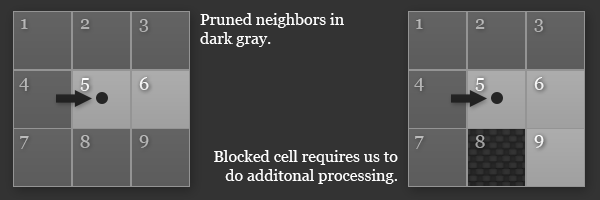
**d)** It is important to have admissible heuristic in A\* to ensure optimality because if we don’t have admissible heuristic then we will have an estimation that is bigger than the actual path cost from some initial state to a goal state. And if this high cost path estimation is on the least cost path (or optimal path that we wanted) then our algorithm by default will not explore it because it seems more expensive. Then our algorithm will go towards another path (which will clearly not be the most optimal)

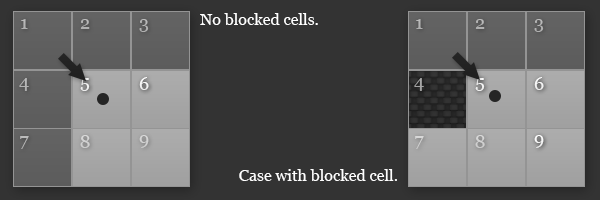
For example:

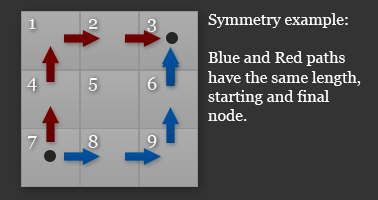


Let node ‘A’ be the initial state and node ‘G’ be the goal state.   
We will have a heuristic function h(N) for any N state of the above graph which gives as an estimate to the goal state. We will also have a have a cost function c(N,M) which is the step cost of any node N with its neighbor M.  
  
We will assume some heuristic values which won’t be admissible to test things out.  
Let h(B)=3 and h(C)=4. This is not admissible since we are overestimating the value as h(C) > c(C,G) since the step cost from C to G is 2 ( as seen from the graph.  
  
Now,  
If the A\* algorithm runs on this then from node A it will go to node B instead of C and then will reach the Goal. The path that it will choose will not be the optimal path (which is clear from our graph). It didn’t choose the most optimal path because of the fact that we didn’t make our heuristic admissible.

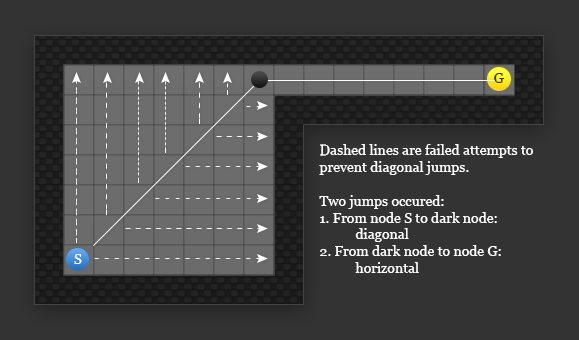
**e)** A\* is a very efficient algorithm for pathfinding. As we need pathfinding algorithms to be more and more fast as well as memory efficient A\* algorithm variants of been proposed one of which is to use ‘jump point search algorithm’ with A\* algorithm so that the speed of A\* algorithm can be increased. In any pathfinding algorithm there will be a big relation between the performance of the algorithm and the size of the graph that is being searched. What Jump Point Search really does is that it skips some of nodes which we by default needed to add in open list and closed list (form A\*) to eliminate a lot of intermediate nodes in certain kind of combination and calculations. In a typical A\* algorithm we start with a node and explore its neighbors. Then we pick the node which has the least value of f(n). In this variant example instead of picking adjacent nodes we will use a function to do it for us. What this does is eliminate nodes that are not interesting to our path.   
  
In order for us to understand the elimination of nodes we take an example of pruning the nodes which we will ignore or eliminate for horizontal and vertical directions:  


(Example of one of horizontal pruning situations.)

  
(Example of diagonal pruning situations.)  
  
When we have picked a neighbor using our function we try to find a jump point to the new node from our current node. This node can be reached from the current node in more than just one way. Basically what JPS does is it eliminates symmetry between paths.

  
(Path symmetry example)  
  
This elimination of symmetry looks futile in such a small grid but when we have a big open space where there will be a lot of possibilities to explore this can result as a huge optimization and increases the general speed of the algorithm

So for big open spaces we can have huge wins.

  
(Example of jump function behavior.)  
  
In the diagonal case we look at all the nodes in the horizontal and vertical direction and if those fail (like they have in the above graph) then we have to have to put a forced node as a jump point so we can reach their from our current state and then reach the goal state.

If we don’t find a jump node we will keep on calling the jump functions recursively in the specified direction.  
  
The rest of the algorithm works as it is and A\* algorithm then explores the new node that was obtained by the jump function till it reaches the goal node. So in this way this variant A\* algorithm using jump search increases the speed of thee algorithm.

References:  
  
1. https://harablog.wordpress.com/2011/09/07/jump-point-search/

2. Podhraski, T. (2013). How to speed up A\* pathfinding with the jump point search Algorithm. 2013-11-20]. http://gamedev. tutsplus. com/tutorials/implementation/speed-up-a-star-pathfinding-withthe-jump-point-search-algorithm.