<2T>

# A\* Description

A\* search (pronounced a-star search) is an algorithm widely used in pathfinding and graph traversal, the process of plotting an efficiently directed path between multiple points, called "nodes". It enjoys widespread use due to its performance and accuracy. However, in practical travel routing systems, it is generally outperformed by algorithms which can preprocess the graph to attain better performance, although other work has found A\* to be superior to other approaches. It is an extension of Edsger Dijkstra’s 1959 algorithm. A\* achieves better performance by using heuristics to guide its search.

A\* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance travelled, shortest time, etc.), and among these paths it first considers the ones that *appear* to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node.

At each iteration of its main loop, A\* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A\* selects the path that minimizes

{\displaystyle f(n)=g(n)+h(n)}

where *n* is the last node on the path, *g*(*n*) is the cost of the path from the start node to *n*, and *h*(*n*) is a heuristic that estimates the cost of the cheapest path from *n* to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

# A\* Pseudo-code

**function** A\*(start, goal)

*// The set of nodes already evaluated*

closedSet := *{}*

*// The set of currently discovered nodes that are not evaluated yet.*

*// Initially, only the start node is known.*

openSet := *{start}*

*// For each node, which node it can most efficiently be reached from.*

*// If a node can be reached from many nodes, cameFrom will eventually contain the*

*// most efficient previous step.*

cameFrom := an empty map

*// For each node, the cost of getting from the start node to that node.*

gScore := map **with** default value **of** Infinity

*// The cost of going from start to start is zero.*

gScore[start] := 0

*// For each node, the total cost of getting from the start node to the goal*

*// by passing by that node. That value is partly known, partly heuristic.*

fScore := map **with** default value **of** Infinity

*// For the first node, that value is completely heuristic.*

fScore[start] := heuristic\_cost\_estimate(start, goal)

**while** openSet **is** **not** empty

current := the node **in** openSet having the lowest fScore[] value

**if** current = goal

return reconstruct\_path(cameFrom, current)

openSet.Remove(current)

closedSet.Add(current)

**for** each neighbor **of** current

**if** neighbor **in** closedSet

**continue** *// Ignore the neighbor which is already evaluated.*

**if** neighbor **not** **in** openSet *// Discover a new node*

openSet.Add(neighbor)

*// The distance from start to a neighbor*

*//the "dist\_between" function may vary as per the solution requirements.*

tentative\_gScore := gScore[current] + dist\_between(current, neighbor)

**if** tentative\_gScore >= gScore[neighbor]

**continue** *// This is not a better path.*

*// This path is the best until now. Record it!*

cameFrom[neighbor] := current

gScore[neighbor] := tentative\_gScore

fScore[neighbor] := gScore[neighbor] + heuristic\_cost\_estimate(neighbor, goal)

return failure

**function** reconstruct\_path(cameFrom, current)

total\_path := [current]

**while** current **in** cameFrom.Keys:

current := cameFrom[current]

total\_path.append(current)

return total\_path

# A\* Code

import heapq

#Simple priority queue class

class PQueue:

def \_\_init\_\_(self):

self.elements = []

def is\_empty(self):

return len(self.elements) == 0

def put(self, item, priority):

heapq.heappush(self.elements, (priority, item))

def get(self):

return heapq.heappop(self.elements)[1]

#Replace with better admissable heuristic fuction for your purpose

def heurisitc(x,y):

return 1

def a\_star(graph, start, goal):

discovered = PQueue()

discovered.put(start,0)

parent = {}

cost = {}

parent[start] = None

cost[start] = 0

while not discovered.is\_empty():

current = discovered.get()

if current == goal:

break

for next in graph.neighbors(current):

new\_cost = cost[current] + graph.cost(current, next)

if next not in cost or new\_cost < cost[next]:

cost[next] = new\_cost

priority = new\_cost + heuristic(goal, next)

discovered.put(next, priority)

parent[next] = current

return patent, cost