1. **Requirements Documentation** 
   1. **Description of Problem**

**Name:** Create an A\*star class that implements the A\*star algorithm and provide an application that visually displays the visual effects.

**Problem Statement:** (Create a program that implements the A\*star algorithm.)

**Problem Specifications:** (Create a program that correctly implements the A\*star error free)

* 1. **Input Information**

(Through the PyGame window and certain inputs controls on the keyboard and mouse.)

* 1. **Output Information**

(The quickest and less weighted path from the start node to the goal node.)

(The G score, H score, and F scores of certain nodes.)

(The open list, closed list, and neighbors for the graph.)

* 1. **User Interface**

(The info being displayed is G, H, F scores, neighboring nodes, open list, closed list and the path between the starting and goal nodes.)

1. **System Architecture**

**Vector2.py**

**Prototype:** (def add (self, other)

**Description:** (A function to add two vect2s.)

**Precondition:** (2 different Vectors)

**Post condition:** (the sum of the vect2s is returned.)

**Protection Level:** (public)

**Prototype:** (def sub (self, other)

**Description:** (A function that subtracts two vect2s.)

**Precondition:** (2 different vectors.)

**Post condition:** (The remainder of the two vectors is returned.)

**Protection Level:** (public)

**Prototype:** (def multiply by scalar (self, other)

**Description:** (A function that multiplies the vector by a scalar.)

**Precondition:** (The product of the vector and scalar is returned.)

**Post condition:** (public)

**Protection Level:** (Protection level of the function with in the class

**Prototype:** (def dot (self, other)

**Description:** (A function that returns the dot product of two vectors.)

**Precondition:** (Two different Vectors.)

**Post condition:** (The dot product is returned.)

**Protection Level:** (Public)

**Prototype:** (def magnitude (self)

**Description:** (A function that returns the magnitude of a vector.)

**Precondition:** (A Vector)

**Post condition:** (The magnitude of a vector is returned.)

**Protection Level:** (Public)

**Prototype:** (def normalize (self)

**Description:** (returns a vector normalized)

**Precondition:** (A vector)

**Post condition:** (the vector normalized is returned.)

**Protection Level:** (Public)

**Prototype:** (def E Q(self, other)

**Description:** (A function that overloads the comparison operator.)

**Precondition:** (Two vectors to compare)

**Post condition:** (which vector is better is then returned.)

**Protection Level:** (Public)

**A\_Star.py**

**Prototype:** (def sort \_ open \_ list(self)

**Description:** (sorts the open list by F scores.)

**Precondition:** (The F scores of nodes)

**Post condition:** (The open list is sorted by the f scores of nodes)

**Protection Level:** (Public)

**Prototype:** (def A star(self, start \_ node, goal \_ node, A star \_ graph)

**Description:** (A function that loops through the A star algorithm until a path to the goal node is found.)

**Precondition:** (A start node, goal node, and A star \_ graph)

**Post condition:** (the path for the start node to the goal node and parent nodes are returned.)

**Protection Level:** (Public)

**Prototype:** (def draw \_ path(self)

**Description:** (Returns a list of parent nodes that is the path between the start and goal node.)

**Precondition:** (Graph, start node goal node)

**Post condition:** (A path of parent nodes is returned.)

**Protection Level:** (Public)

**GraphAndNodeClass.py**

**Prototype** (def find neighbors (self, node, Graph)

**Description:** (returns a list of neighboring nodes for each new parent in the path.)

**Precondition:** (A Node and Graph.)

**Post condition:** (Returns the neighbors of the parent nodes)

**Protection Level:** (Public)

**Prototype:** (def set parent (self, other)

**Description:** (attempts to change the parent variable.)

**Precondition:** (f score)

**Post condition:** (the parent node is change.)

**Protection Level:** (public)

**Prototype:** (calculate G score (self, other)

**Description:** (calculates the estimated movement cost to move from parent node )

**Precondition:** (x position and y position.)

**Post condition:** (returns the G score.)

**Protection Level:** (Public)

**Prototype:** (calculate H score (self, other)

**Description:** (calculates the estimated movement cost to move from this node to the goal node.)

**Precondition:** (x position and y position.)

**Post condition:** (returns the h score.)

**Protection Level:** (Public)

**Prototype:** (calculate F score (self, other)

**Description:** (calculates the f score which is the sum of the G and H score)

**Precondition:** (G and H scores)

**Post condition:** (returns the f score.)

**Protection Level:** (Public)

**Prototype:** (set not transversalble (self)

**Description:** (Makes certain nodes non transversable.)

**Precondition:** (x position and y position.)

**Post condition:** (returns the G score.)

**Protection Level:** (Public)

1. **Source Code**
2. '''A class for the node needed in Astar.'''
3. from vector2 import Vector2
4. from GraphAndNodeClass import Node
5. from GraphAndNodeClass import Graph
6. class Astar:
7. def \_\_init\_\_(self):
8. self.Astar\_graph = None
9. self.closed\_list = []
10. self.open\_list = []
11. self.goal\_node = None
12. self.start\_node = None
13. self.paths = []
14. self.current\_node = None
16. def sort\_open\_list(self):
17. '''A function that sorts the open list using elements of the Astar Algorithim.'''
18. #sorts open list by fscore
19. #compare the value in the list to every other putting the smaller value
20. for i in range(0,len(self.open\_list)):
21. for j in range(0,len(self.open\_list)):
22. #Compare the fscore at index i to fscore at index j
23. #if at index j less than index i swap places
24. if (self.open\_list[i].f\_score < self.open\_list[j].f\_score):
25. temp = self.open\_list[i]
26. self.open\_list[i] = self.open\_list[j]
27. self.open\_list[j] = temp
28. def A\_star(self,start\_node,goal\_node, Astar\_graph):
29. '''A function that loops and does the astar algorithim until a path to the goal node is found.'''
30. self.start\_node = start\_node
31. self.goal\_node = goal\_node
32. current = self.start\_node
33. #step1 add starting node to the open list.
34. self.open\_list.append(current)
35. #step2 start Loop while open list is not empty.
36. while self.open\_list:
37. #2a Look for lowest f score in open list.
38. self.sort\_open\_list()
39. current = self.open\_list[0]
40. # 2b switch the lowest f score to the closed list.
41. self.open\_list.remove(current)
42. self.closed\_list.append(current)
43. if self.closed\_list.\_\_contains\_\_(self.goal\_node):
44. return self.draw\_path()
45. # 2c Get neighbors.
46. temp = self.find\_neighbors(current, Astar\_graph)
47. # 2d Loop through all neighbors.
48. for node in temp:
49. #if non transvisble or in closed list
50. if not node.traversable or self.closed\_list.\_\_contains\_\_(node):
51. continue
52. #otherwise if not in the openlist
53. if not self.open\_list.\_\_contains\_\_(node):
54. self.open\_list.append(node)
55. #then calculate g,h,f, and score
56. node.calculate\_g\_score(current)
57. node.calculate\_h\_score(goal\_node)
58. node.calculate\_f\_score()
59. #add to open list
60. #if in the open list check path by compareing g scores
61. #if in closed list break out of the Loop
62. if self.closed\_list.\_\_contains\_\_(goal\_node):
63. break
64. #if goal in closed list return path
65. def draw\_path(self):
66. path = []
67. current = self.goal\_node
68. while current is not None:
69. path.append(current)
70. current = current.parent
71. return path
73. def find\_neighbors(self, node, graph):
74. valid\_neighbors = []
75. top = (node.position + Vector2(0, 1)) #Top\_neighbor
76. bottom = (node.position + Vector2(0, -1)) #Bottom\_neighbor
77. left = (node.position + Vector2(-1, 0)) #Left\_neighbor
78. right = (node.position + Vector2(1, 0)) #Right\_neighbor
79. top\_right = (node.position + Vector2(1, 1)) #Top Right\_neighbor
80. top\_left = (node.position + Vector2(-1, 1)) #Top Left\_neighbor
81. bottom\_right = (node.position + Vector2(1, -1)) #Bottom Right\_neighbor
82. bottom\_left = (node.position + Vector2(-1, -1)) #Bottom Left\_neighbor
83. valid\_neighbors.append(top)
84. valid\_neighbors.append(left)
85. valid\_neighbors.append(right)
86. valid\_neighbors.append(bottom)
87. valid\_neighbors.append(top\_left)
88. valid\_neighbors.append(top\_right)
89. valid\_neighbors.append(bottom\_left)
90. valid\_neighbors.append(bottom\_right)
92. neighbors = []
93. for n in graph:
94. for pos in valid\_neighbors:
95. if n.position == pos:
96. neighbors.append(n)
97. return neighbors
98. def main():
99. b = Graph(10)
100. a = Astar()
101. path = a.A\_star(b.nodes[0], b.nodes[99], b)
102. a = 0
103. from vector2 import Vector2
104. class Graph(object):
105. '''A class that defines the properties of a graph.'''
106. def \_\_init\_\_(self, dims):
107. self.width = dims
108. self.height = dims
109. self.totalsize = self.width \* self.height
110. self.nodes = []
111. id = 0
112. for i in range(0, self.height):
113. for y in range(0, self.width):
114. n = Node(Vector2(i,y))
115. self.nodes.append(n)
117. class Node(object):
118. '''A class created to define the properties of a node.'''
119. def \_\_init\_\_(self, pos):
120. self.position = pos
121. self.g\_score = 0
122. self.h\_score = 0
123. self.f\_score = 0
124. self.parent = None
125. self.is\_traversable = True
126. self.neighbors = None
127. self.is\_goal = False
128. self.is\_start = False
129. def find\_neighbors(self,graph):
130. '''Gets the neighbors of the node. Used to generate the correct path for the user to
131. test against'''
132. valid\_neighbors = []
133. top = (self.position + Vector2(0, 1)) #Top\_neighbor
134. bottom = (self.position + Vector2(0, -1)) #Bottom\_neighbor
135. left = (self.position + Vector2(-1, 0)) #Left\_neighbor
136. right = (self.position + Vector2(1, 0)) #Right\_neighbor
137. top\_right = (self.position + Vector2(1, 1)) #Top Right\_neighbor
138. top\_left = (self.position + Vector2(-1, 1)) #Top Left\_neighbor
139. bottom\_right = (self.position + Vector2(1, -1)) #Bottom Right\_neighbor
140. bottom\_left = (self.position + Vector2(-1, -1)) #Bottom Left\_neighbor
141. valid\_neighbors.append(top)
142. valid\_neighbors.append(left)
143. valid\_neighbors.append(right)
144. valid\_neighbors.append(bottom)
145. valid\_neighbors.append(top\_left)
146. valid\_neighbors.append(top\_right)
147. valid\_neighbors.append(bottom\_left)
148. valid\_neighbors.append(bottom\_right)
150. neighbors = []
151. for n in graph.nodes:
152. for pos in valid\_neighbors:
153. if n.position == pos:
154. neighbors.append(n)
155. return neighbors
156. def set\_parent(self, other):
157. '''Attempts to change the value of the parent variable. If the value is
158. None the parent is automaticly set to the value of node passed in. Otherwise
159. a new G score is calcualted and if it is cheaper than the current the parent
160. is changed to the node passed in and the G score is modified to reflect the
161. change in parents'''
162. self.parent = other
163. return self.parent
165. def calculate\_g\_score(self, other):
166. '''Calculates the movement cost to move from nodes parent to it self. If the
167. movement is horizontal or vertical the cost is parent's G score + 10. If the
168. movement is diagonal the cost is parent's G score + 14'''
169. if self.parent is None:
170. if ((self.position.xpos is other.position.xpos and self.position.ypos is not other.position.ypos)
171. or (self.position.xpos is not other.position.xpos and self.position.ypos is other.position.ypos)):
172. self.g\_score = other.g\_score + 10
173. else:
174. self.g\_score = other.g\_score + 14
175. self.set\_parent(other)
176. elif self.parent is not None:
177. tentative\_g = self.g\_score
178. if ((self.position.xpos is other.position.xpos and self.position.ypos is not other.position.ypos)
179. or (self.position.xpos is not other.position.xpos and self.position.ypos is other.position.ypos)):
180. tentative\_g = other.g\_score + 10
181. else:
182. tentative\_g = other.g\_score + 14
183. if tentative\_g < self.g\_score:
184. self.g\_score = tentative\_g
185. self.set\_parent(other)
186. def calculate\_h\_score(self, other):
187. '''Calculates the estimated movement cost to move from this node to the goal.'''
188. x\_distance = abs(other.position.xpos - self.position.xpos)
189. y\_distance = abs(other.position.ypos - self.position.ypos)
190. total = x\_distance + y\_distance
191. self.h\_score = total \* 10
192. return self.h\_score
193. def calculate\_f\_score(self):
194. '''Calculates the fscore which is the sum of the H score and G score of the node'''
195. self.f\_score = self.g\_score + self.h\_score
196. return self.f\_score
197. def set\_not\_traversable(self):
198. '''Makes a certain selected spot nontransversible.'''
199. self.is\_traversable = False
200. def main():
201. g = Graph(10)
202. a = g.nodes[0].find\_neighbors(g)
203. a[0].calculate\_g\_score(g.nodes[0])
204. a[1].calculate\_g\_score(g.nodes[0])
205. a[2].calculate\_g\_score(g.nodes[0])
206. b = 0;

**Read Me**

(Be very clear as to how the assessor should go about getting your application, running it, and using it. You should assume the assessor knows nothing about your application.)