((گزارش کار پروژه اول هوش مصنوعی))

گرداورنده : عرفان ماجدی 9831099

دکتر جوانمردی

پاسخ سوال اول)

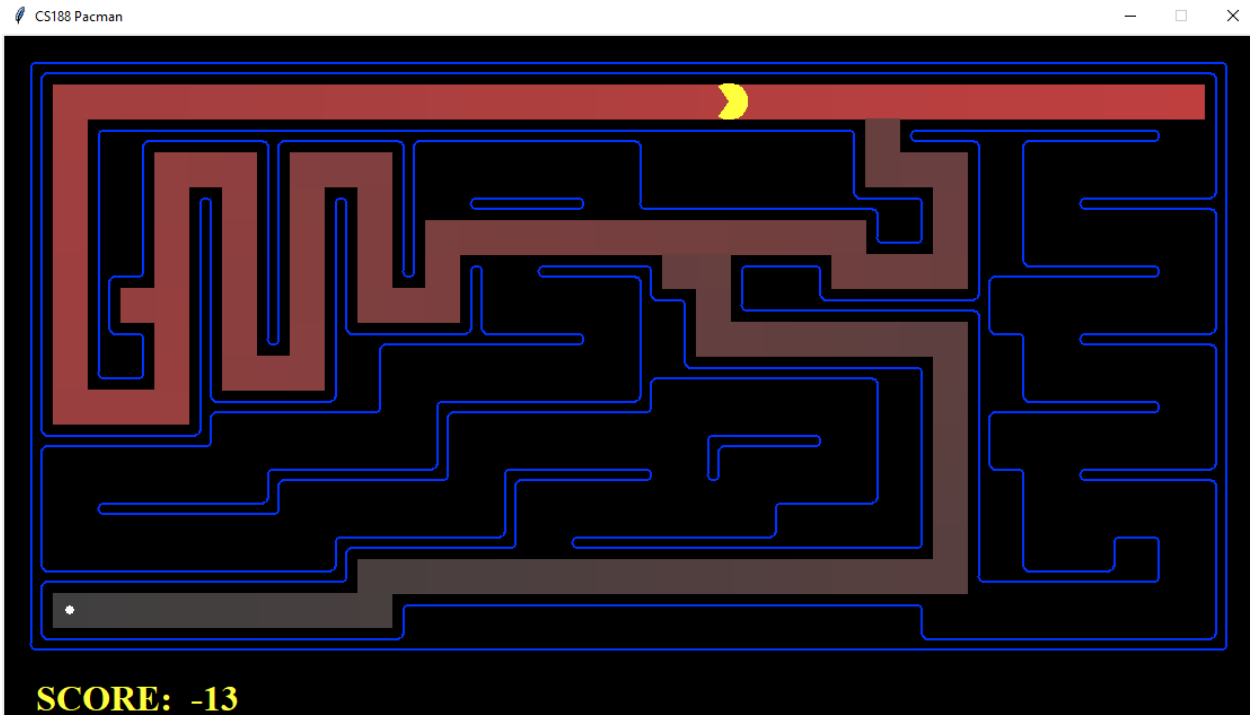
بله ترتیب مورد انتظار است و همچنین به خاطر الگوریتمی که DFS دارد از همه ی مربع های کاوش شده نمی گذرد .

پاسخ سوال دوم)

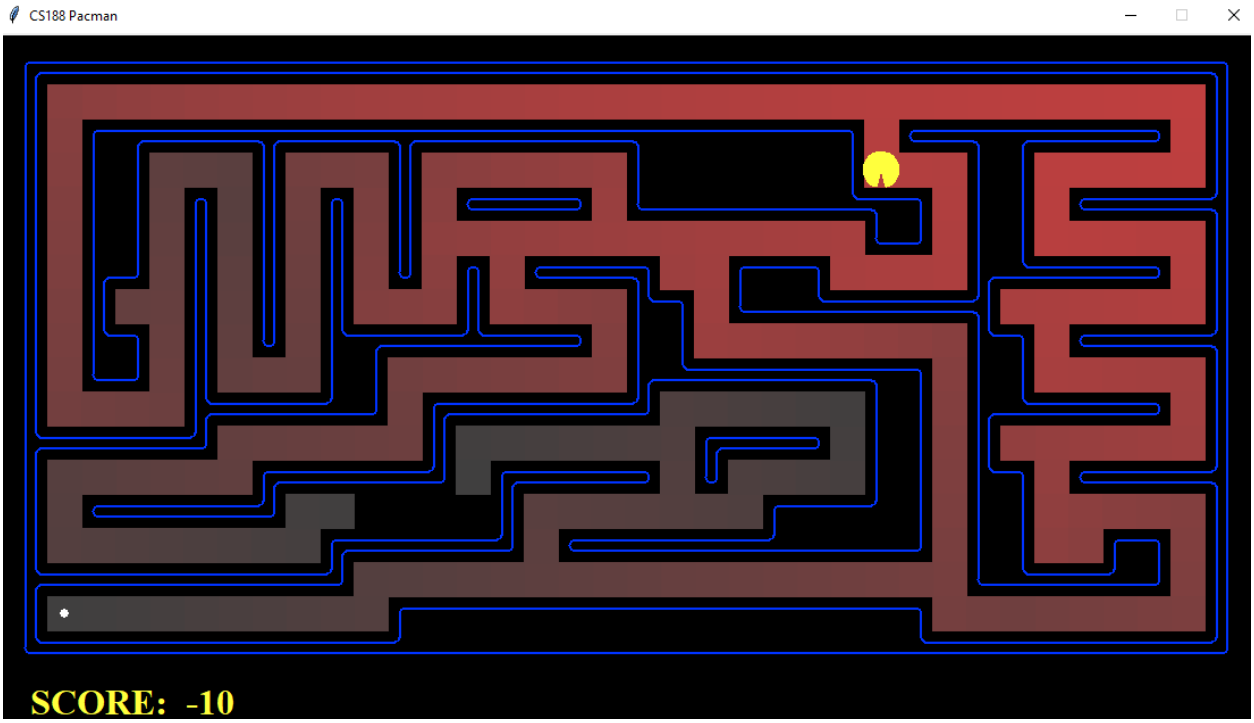
خیر زیرا در جست و جوی DFS ممکن است جوابی وجود داشته باشد که در چپ ترین node باشد اما در اخرین عمق نباشد به همین دلیل لزوما بهینه نیست پس مشکل آن این است که ممکن است جواب به عنوان مثال در راست ترین node باشد ولی این الگوریتم هزینه ی زیادی را ایجاد میکند تا به آن برسد .

پاسخ سوال سوم)

برای توضیح این بخش به عکس های زیر توجه کنید :

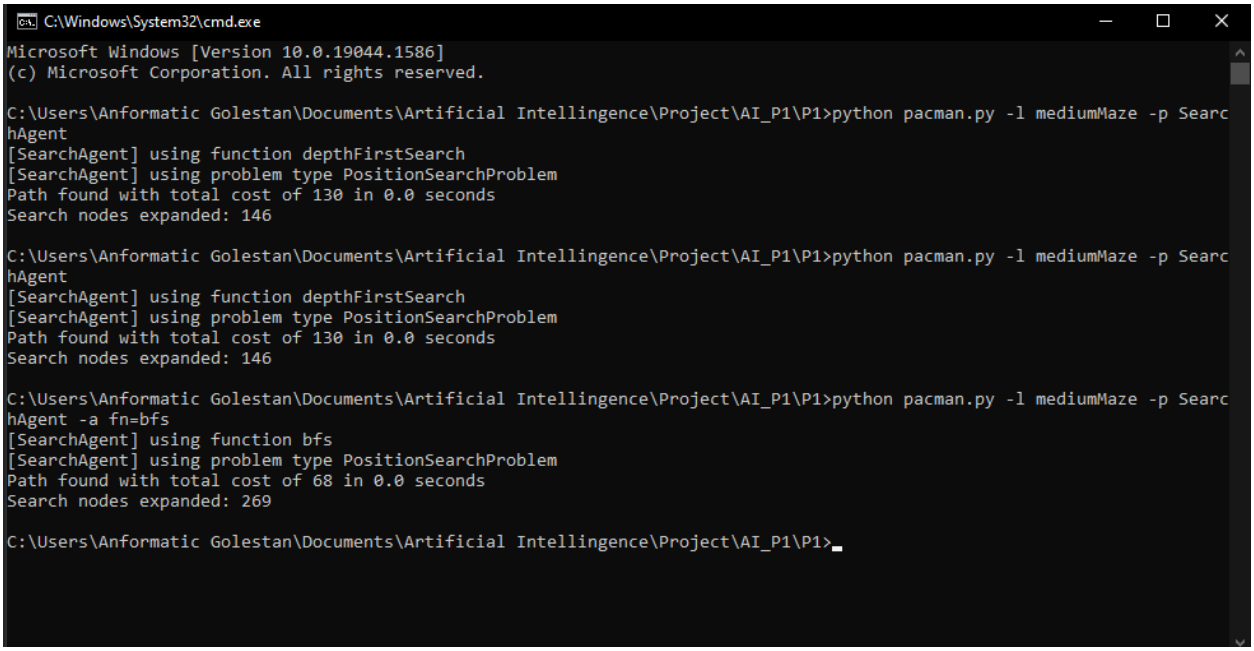


Pacman in dfs medium maze



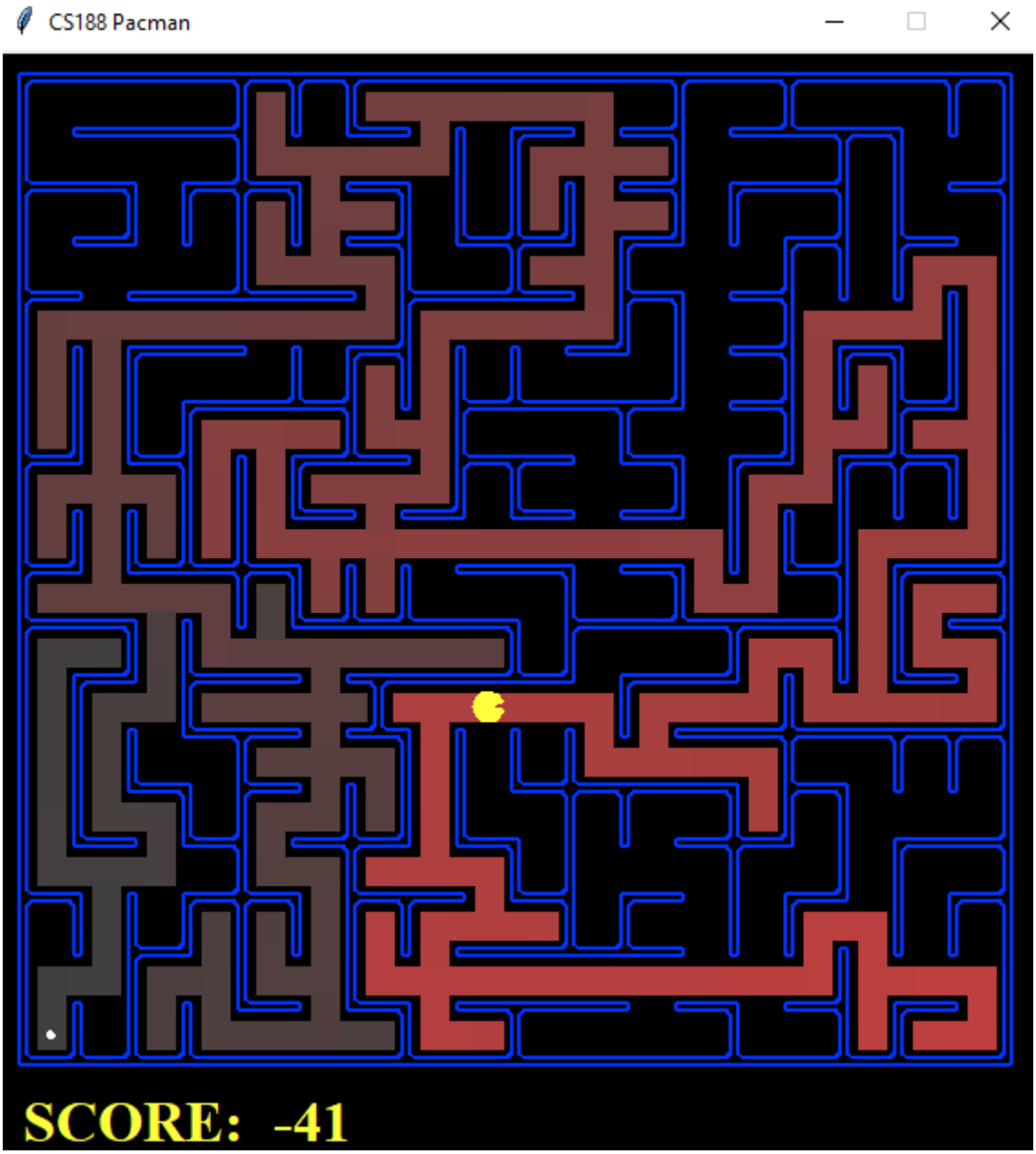
Pacman in bfs medium maze

با توجه به شکل بالا چون نقطه در چپ ترین نقطه قرار دارد پس الگوریتم DFS به خاطر سبک الگوریتمش node های کمتری را گسترش می دهد پس اولین دلیل تفاوت بین این DFS و BFS می تواند به خاطر الگوریتم و سبک جست و جوی آن ها باشد . به عکس زیر هم پس از اجرا توجه کنید :

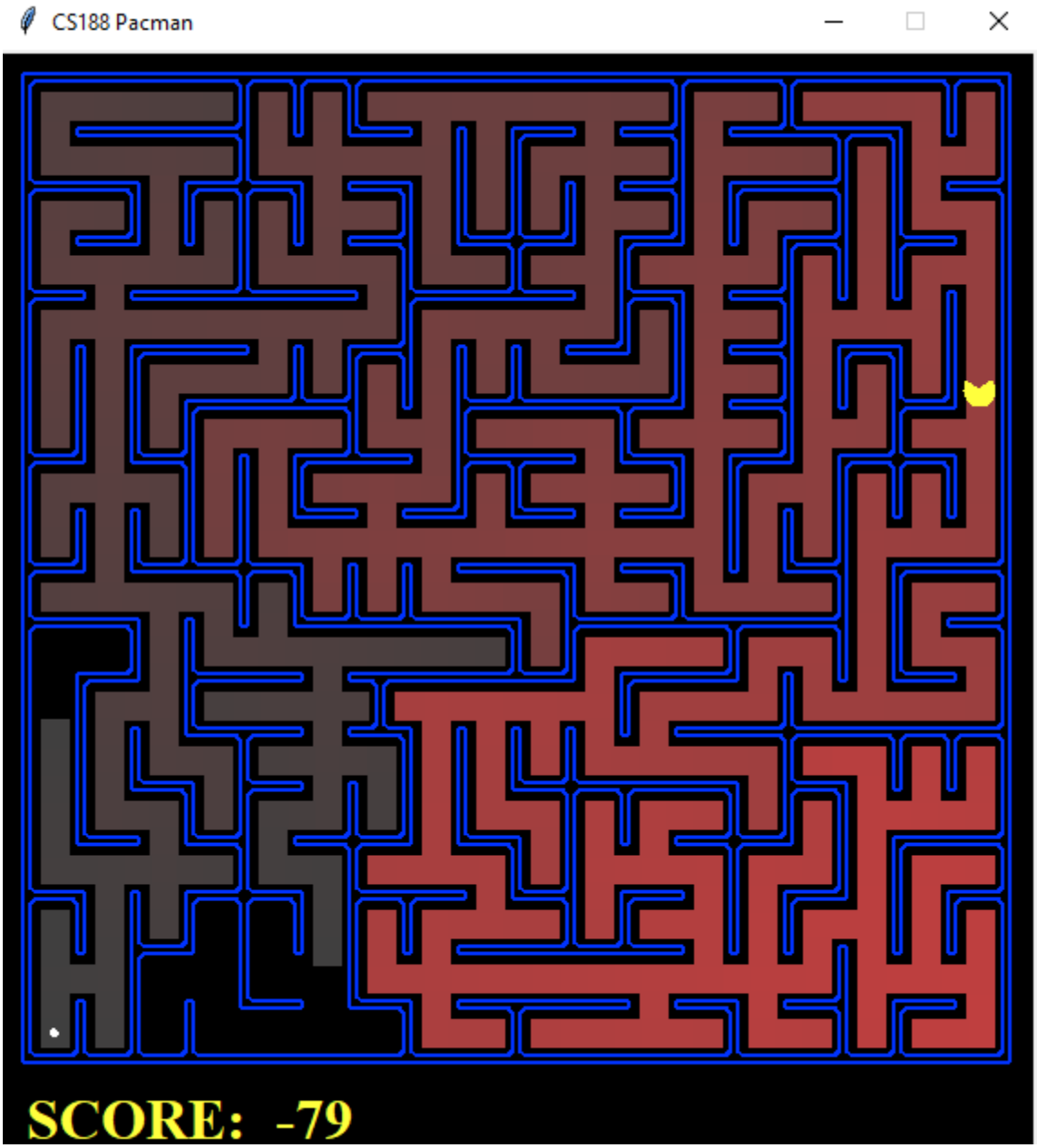


تفاوت بعدی که با توجه کد می توان به آن اشاره کرد به این است که DFS از stack استفاده می کند و BFS از queue استفاده می کند . با توجه به ماهیت stack چون نقطه برای پکمن در چپ ترین نقطه قرار دارد و stack هم به صورت LIFO است و اخرین خانه را اول POP میکند پس با گسترش تعداد node کمتری به هدف می رسد .

توضیحات برای big maze هم به همین شکل است و تصاویر زیر هم گویای ماجرا است :



Pacman in dfs big maze



Pacman in bfs big maze

پاسخ سوال چهارم)

بله ممکن است در صورتی که هزینه ی ما در تمامی action ها برابر یک باشد به BFS می رسیم . به شکل زیر توجه کنید :

```
169  def uniformCostSearch(problem):
170      """Search the node of least total cost first."""
171      start = problem.getStartState()
172      frontier = util.PriorityQueue()
173      explored = []
174      if problem.isGoalState(start) :
175          return []
176
177      frontier.push((start, [], 0),0)
178      while not frontier.isEmpty():
179          state = frontier.pop()
180          if problem.isGoalState(state[0]):
181              return state[1]
182          if state[0] not in explored :
183              explored.append(state[0])
184              successors = problem.getSuccessors(state[0])
185              for successor in successors :
186                  if successor[0] not in explored :
187                      path = state[1] + [successor[1]]
188                      frontier.push((successor[0],path), problem.getCostOfActions(path))
189      return path
```

در خط 188 قسمتی که highlight شده است را اگر پاک کنیم و به جای آن 1 قرار دهیم در واقع تابع هزینه را طوری تغییر دادیم که به BFS می رسیم .

پاسخ سوال پنجم)

$$A^* = \begin{Bmatrix} cost = 54 \\ node\ expanded = 535 \\ score = 456 \end{Bmatrix}$$

$$UCS = \begin{Bmatrix} cost = 54 \\ node\ expanded = 682 \\ score = 456 \end{Bmatrix}$$

$$BFS = \begin{Bmatrix} cost = 54 \\ node\ expanded = 682 \\ score = 456 \end{Bmatrix}$$

$$DFS = \begin{Bmatrix} cost = 298 \\ node\ expanded = 806 \\ score = 212 \end{Bmatrix}$$

با توجه به موارد بالا تفاوتی که الگوریتم های جست و جو در openMaze دارند در تعداد node های expand شده است همچنین الگوریتم DFS بدترین عملکرد را در این maze دارد ولی الگوریتم های جست و جوی A^* و UCS فقط در تعداد node های expand شده تفاوت دارند در حالی که UCS و BFS با هم در این مورد هم برابر هستند .

پاسخ سوال ششم)

در توضیح heuristic می توان گفت که مینیموم فاصله ی corner هایی که explored نشده است را پیدا می کند و با هم دیگر جمع میکند . اگر بخواهیم استدلال کنیم این heuristic ویژگی consistent را رعایت می کند زیرا همیشه گوشه های یکسانی را انتخاب می کند برای یک وضعیت داده شده . (در کامنت کد هم توضیح داده شده است)

پاسخ سوال هفتم)

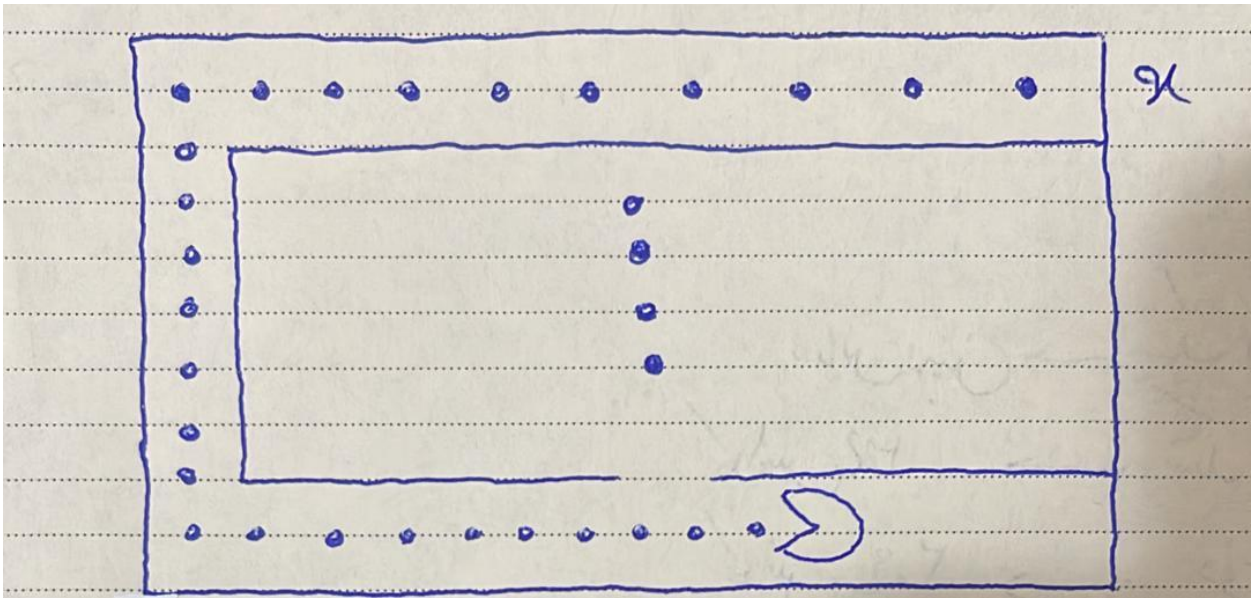
در اینجا از فاصله ی Manhattan و maze استفاده می کنیم . (فاصله ی maze در واقع فاصله ی پکمن تا مقصد با در نظر گرفتن دیوارها است) توضیح heuristic به این صورت است که فاصله ی Manhattan را حساب می کنیم با توجه به صورت مسئله ی پروژه و اینکه حد بالای autograder در این قسمت 5 است پس می گوییم اگر فاصله ی Manhattan از 5 کمتر شد همین فاصله در نظر گرفته شود ولی اگر بیشتر شد از فاصله ی maze استفاده خواهیم کرد. توجه داشته باشید سرعت maze از Manhattan کمتر است ولی اگر مقدار از 5 بیشتر باشد نیاز داریم تا دقت بیشتر باشد و فاصله ی maze این قابلیت را با وجود کندتر بودن دارد.

پاسخ سوال هشتم)

تفاوت این است که در بخش foodHeuristic ما heuristic را طوری طراحی کردیم که طبق یک شرط از فاصله ی Manhattan یا maze استفاده کند ولی در cornerHeuristic فقط نیاز به فاصله ی Manhattan داشتیم . همچنین در cornerHeuristic در واقع کاری که انجام می دادیم روی گوشه هایی بود که هنوز بررسی نشده بودند و پس از بررسی از لیست بررسی نشده ها خارج می شدند ولی در foodHeuristic بررسی های ما روی لیست غذاها بوده و ما فاصله ها را از آن می سنجیدیم یعنی مثلا در state ای که هستیم چقدر فاصله با غذا داریم .

پاسخ سوال نهم)

فرض کنید یک maze ساده داریم مانند شکل زیر :



با توجه به شکل اصولا اگر پکمن optimal حرکت می کرد باید ابتدا بالا برود و سپس به سمت چپ حرکت کند ولی ابتدا تا نقطه ی x می رود و تمام نقاط را خورده و سپس برمیگردد و این سه نقطه را میخورد .