

BFS_Solver

در الگوریتم **BFS** باید به این شکل عمل کنیم که در هر مرحله تمام بچه‌های یک ند را ببینیم (همه ندهای یک سطح را ببینیم) و سپس به سطح بعدی برویم، برای دستیابی به این خواسته به این شکل عمل می‌کنیم که هر عنصری (ندی) را که دیدم آن را به انتهای یک صف اضافه می‌کنیم، و در هر مرحله عنصر اول صف را برابر می‌داریم و ندهای مجاور آن را پیدا کرده و به انتهای صف اضافه می‌کنیم به این شکل ابتدا تمام ندهای سطح ۱ دیده می‌شوند بعد تمام ندهای سطح ۲ و به همین صورت. شکل شماره ۱ این مسئله را نشان می‌دهد.



شکل ۱: اضافه شدن ندها به لیست در BFS

به همین جهت همین طور که در شکل ۲ می‌توان دید از داده ساختار لیست در پایتون به عنوان صف استفاده می‌کنیم و هر بار عنصر را به انتهای لیست اضافه کرده و از سر آن بر می‌داریم. تا جایی پیش می‌رویم که عنصری که برداشته ایم هدف باشد.

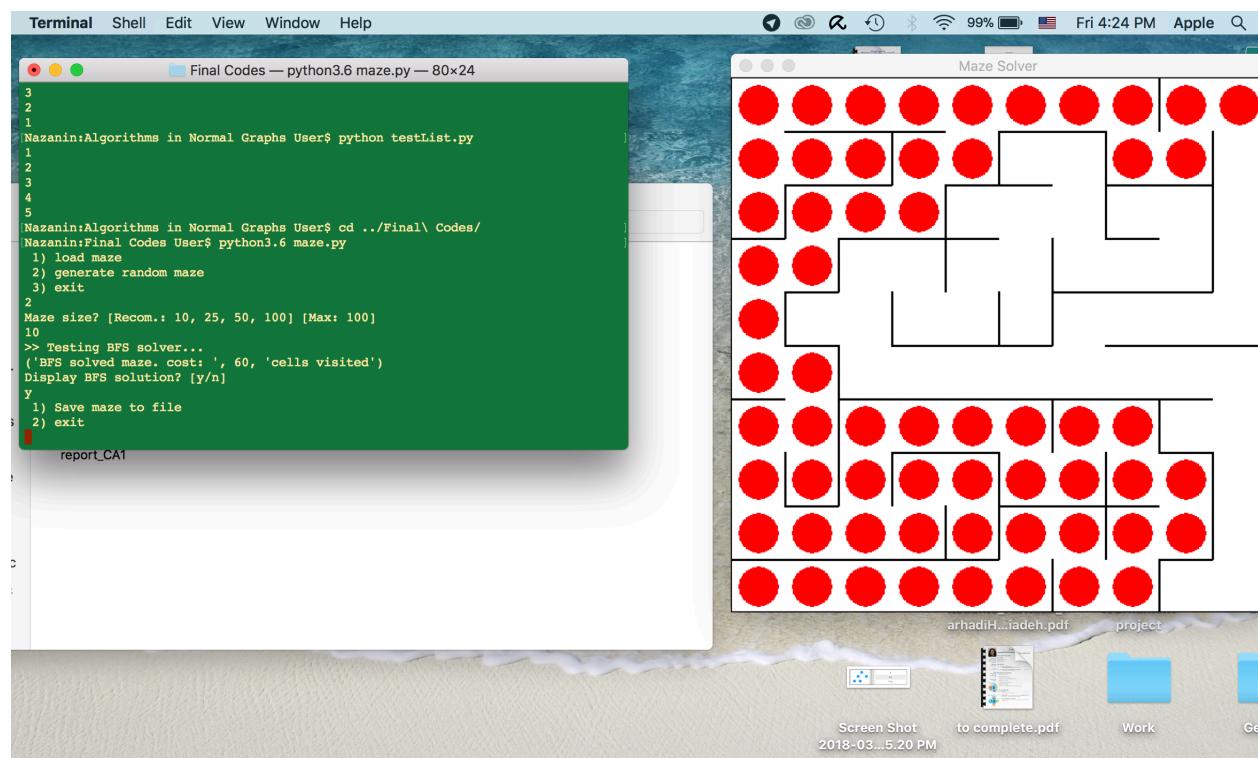
```

138  def bfs_solver(maze):
139      cells = maze.cells
140      if len(cells)==0:
141          print("nothing has been generated")
142      queue = []
143      visited = []
144      for i in range(len(maze.cells)):
145          visited.append([])
146          for j in range(len(maze.cells)):
147              visited[i].append(False)
148      queue.append(maze.start)
149      visited[maze.start[0]][maze.start[1]] = True
150      goal = maze.goal
151      result = []
152      while queue:
153          s = queue.pop(0)
154          result.append(s)
155          if s==goal:
156              break
157          for node in findCellNeigbors(maze, s):
158              if visited[node[0]][node[1]]==False:
159                  queue.append(node)
160                  visited[node[0]][node[1]] = True
161      return result
162

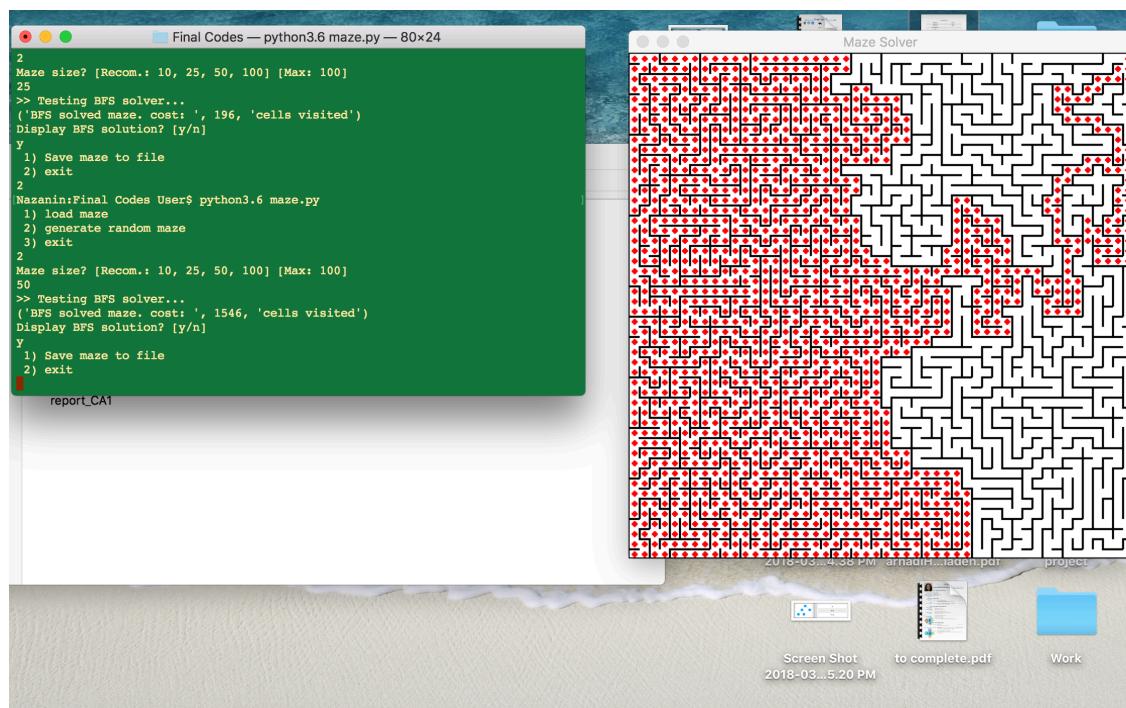
```

شکل ۲: کد اجرای BFS

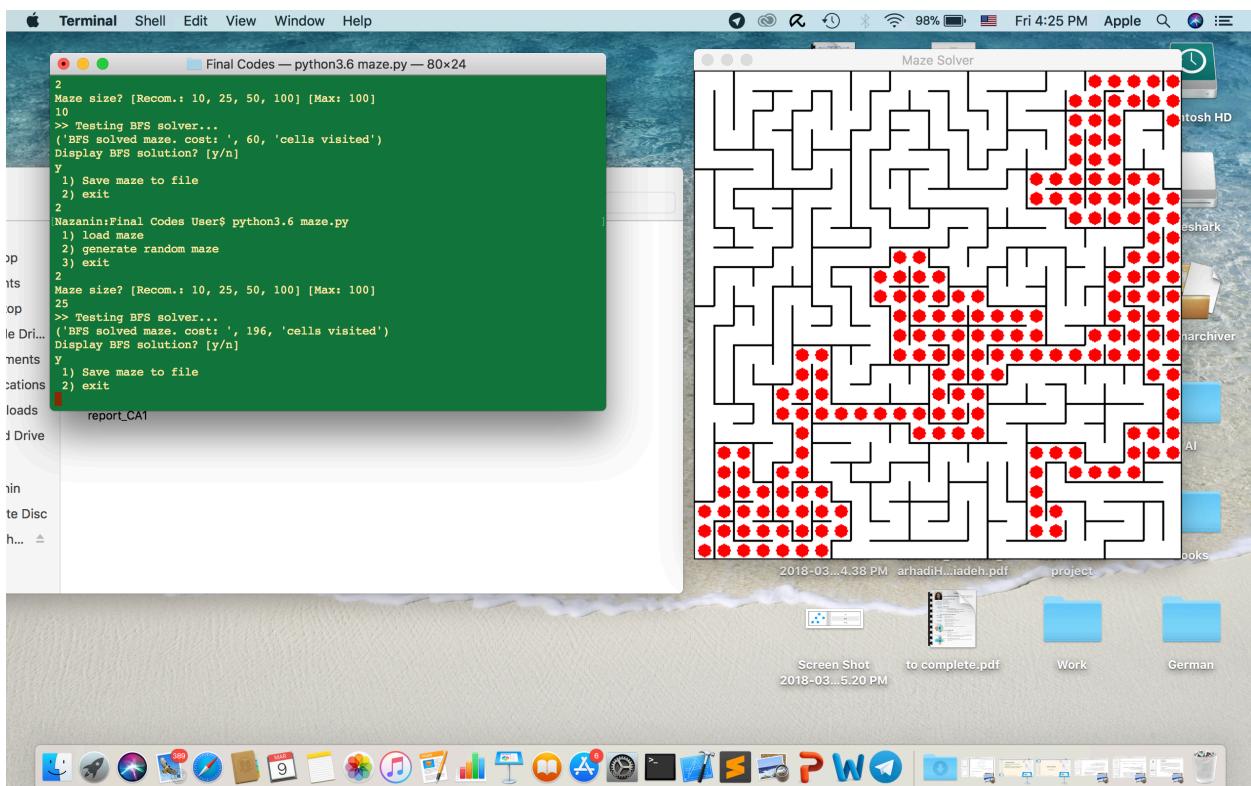
نتایج گرفته شده به ازای اجرای کد برای ابعاد مختلف ماز در شکل‌های ۳، ۴، ۵ و ۶ آمده است.



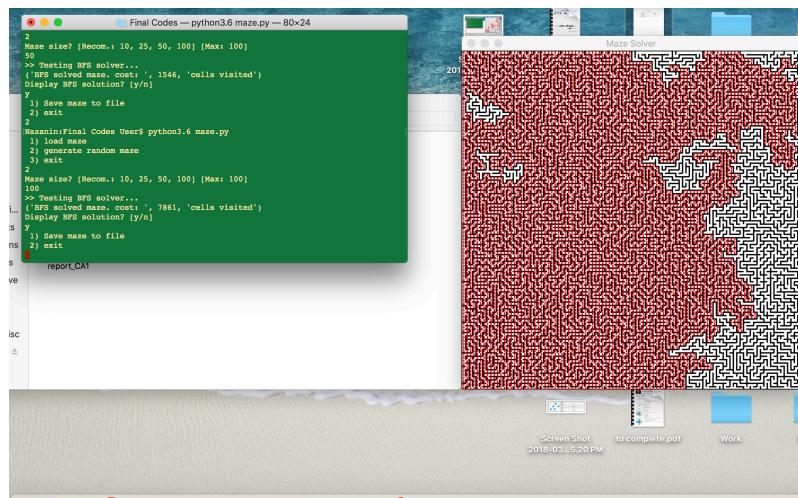
شکل ۳: اجرای BFS برای ماز با اندازه‌ی ۱۰



شکل ۴: اجرای BFS برای ماز با اندازه‌ی ۵۰



شکل ۵: اجرای BFS برای ماز با اندازه‌ی ۲۵



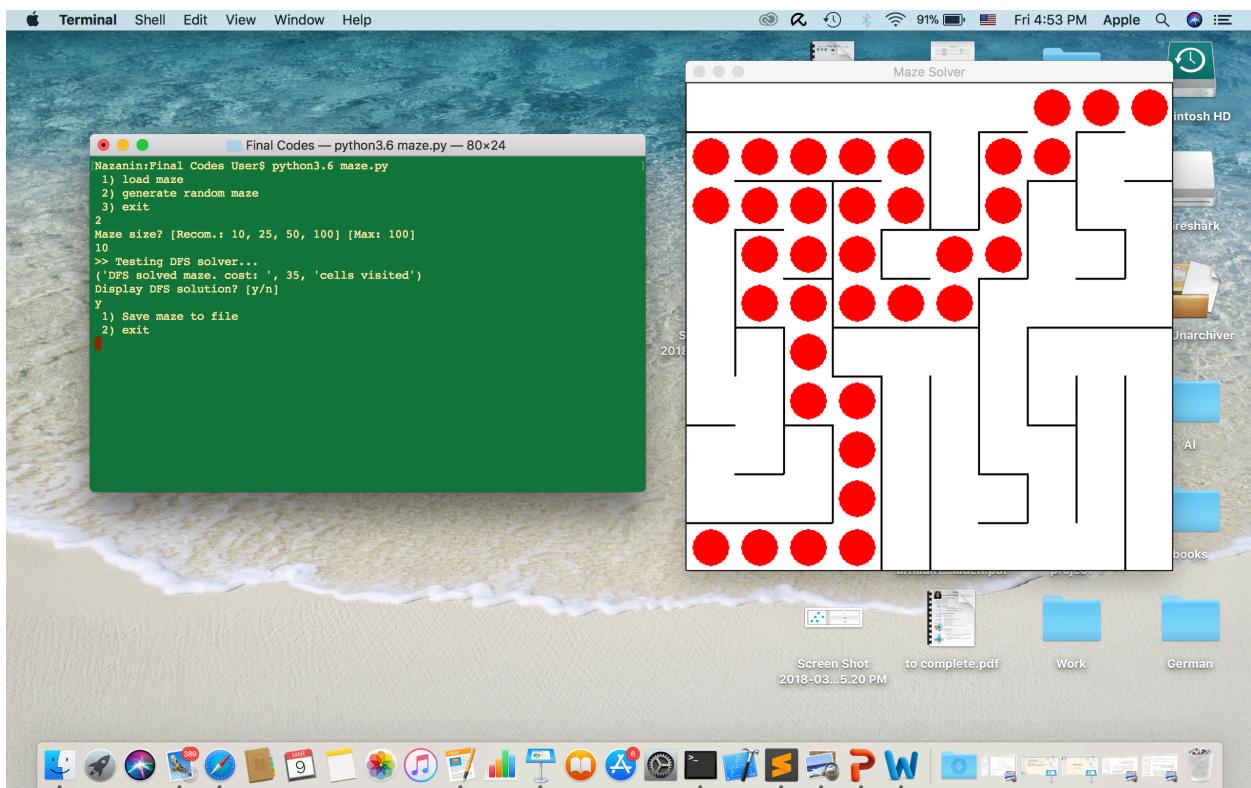
شکل ۶: اجرای BFS برای ماز با اندازه‌ی ۱۰۰

DFS_Solver

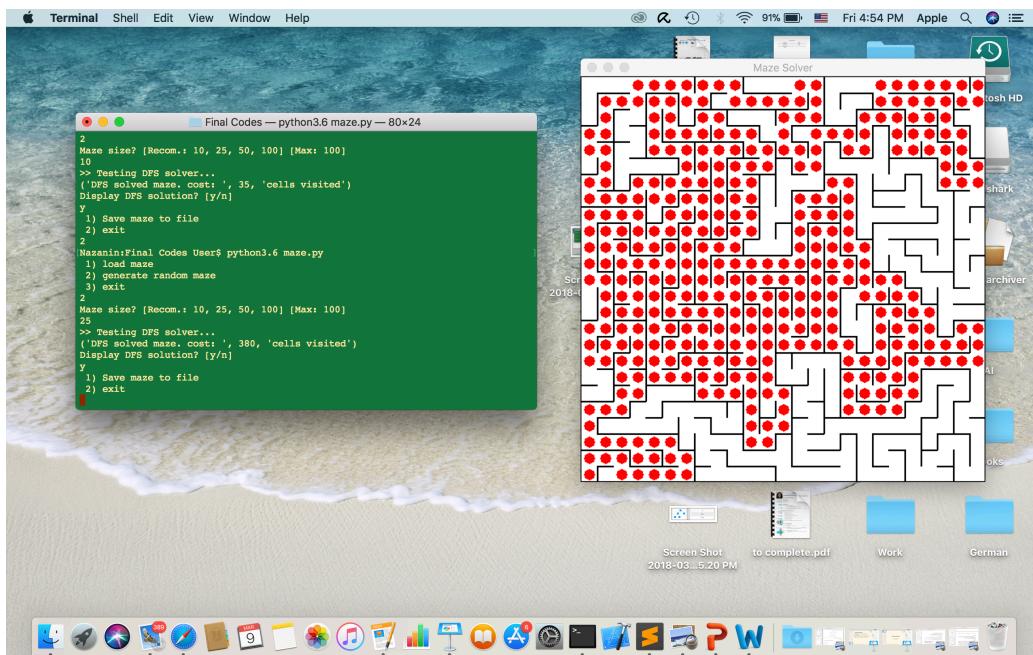
سپس به سراغ الگوریتم DFS می‌روم، در این الگوریتم هر شاخه تا رسیدن به برگ‌هایش پیش می‌رود و اگر هدف در این مسیر نبود برمی‌گردیم و مسیر دیگری را تا رسیدن به برگ ادامه می‌دهیم (در گراف برگ است در این مسئله یعنی تا رسیدن به خانه‌ای که یا به جایی مسیری ندارد یا تمام خانه‌های مجاور آن را قبلاً دیده ایم). ابتدا این الگوریتم را به صورت بازگشتی پیاده سازی کرده بودم اما هنگامی که اندازه‌ی ماز بزرگ می‌شد تعداد دفعات صدا شدن تابع به قدری زیاد بود که ارور می‌داد پر شدن استک تابع را می‌داد به همین دلیل پیاده سازی را با استفاده از داده ساختار استک و به صورت غیر بازگشتی انجام دادم.

علت استفاده از استک در پیاده سازی این بود که در هر مرحله می‌خواهیم بچه‌های یک ند (خانه‌های مجاور آن که قبلاً دیده نشده اند) داخل استک قرار بگیرند و در مرحله‌ی بعد نیز می‌خواهیم یکی از این بچه‌ها را بسط بدیم و نه خواهر یا برادر ند اصلی به همین دلیل استک بهترین داده ساختار برای این نوع پیاده سازی است چون آخرین چیزی که به آن اضافه شده است اولین چیزی است که از آن خارج می‌شود.

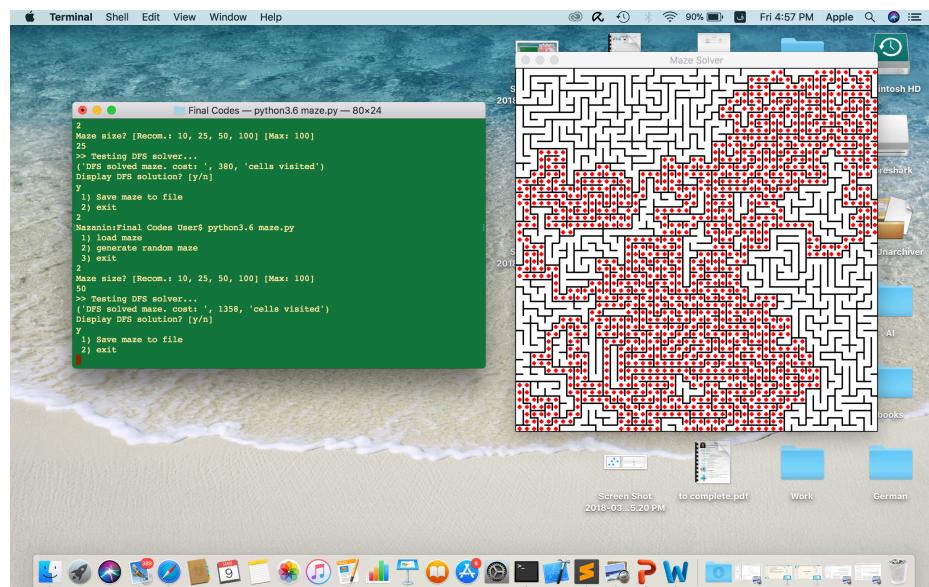
شکل‌های ۷، ۸، ۹ و ۱۰ نتایج اجرای کد این الگوریتم را نشان می‌دهند.



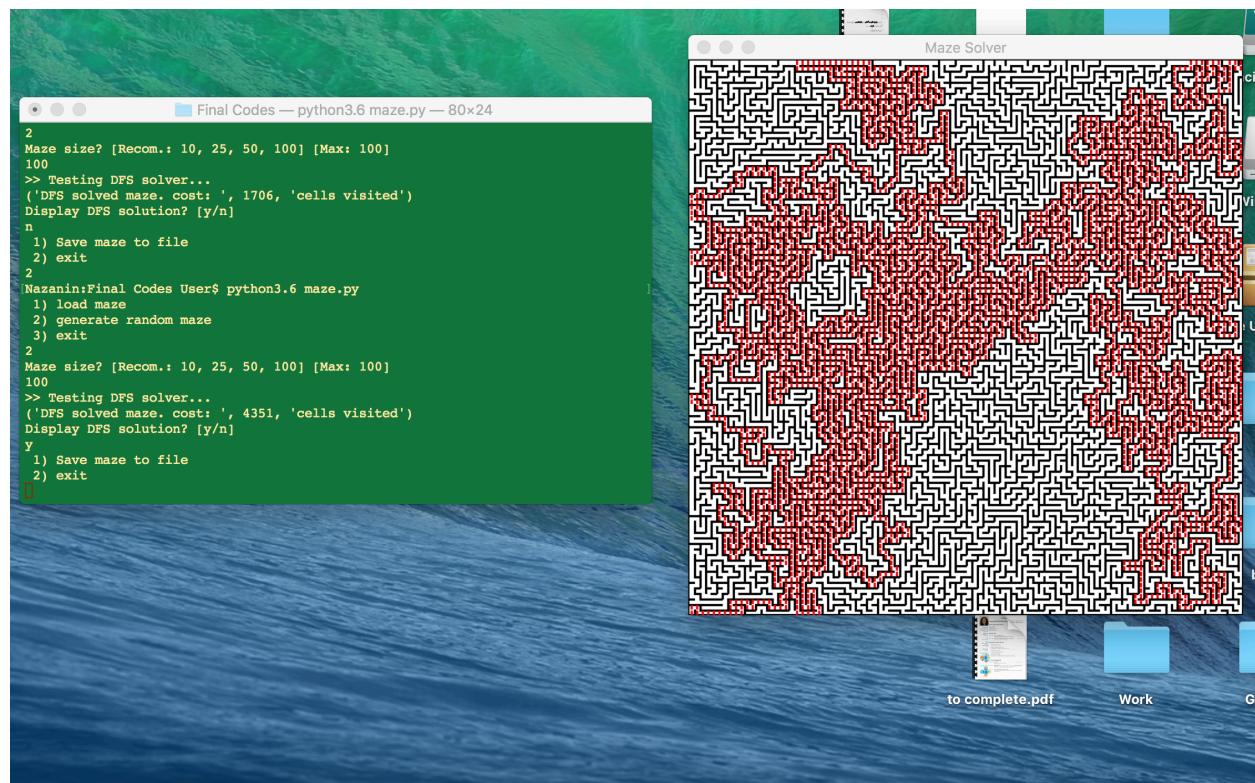
شکل ۷: اجرای DFS برای ماز با اندازه‌ی ۱۰



شکل ۸: اجرای DFS برای ماز با اندازه‌ی ۲۵



شکل ۹: اجرای DFS برای ماز با اندازه‌ی ۱۰



شکل ۱۰: اجرای DFS برای ماز با اندازه ۱۰۰

Iterative_DFS_Solver

همان طور که در شکل ۱۱ دیده می‌شود ایده کلی این روش این است که هر مسیر (شاخه) را تا برگ ادامه ندهیم و فقط تا عمق خاصی ادامه دهیم و این عمق را اندک اندک زیاد کنیم تا بالاخره در یکی از مسیرها هدف هم قرار داشته باشد.

```

79 def iterative_dfs_solver(maze):
80
81     limit = 1
82     while True:
83         dls_path = dls_solver(maze, limit)
84         if maze.goal in dls_path:
85             return dls_path
86         else:
87             dls_solver_result[:] = []
88             limit += 1
89

```

شکل ۱۱: پیاده سازی تابع Iterative_DFS_Solver

برای این کار از DLS کمک می‌گیریم، این تابع نیز در ابتدا به شکل بازگشتی پیاده سازی شده بود ولی به علت مشکل تعداد زیاد صدا شدن تابع پیاده سازی را به روش غیر بازگشتی تغییر دادم.

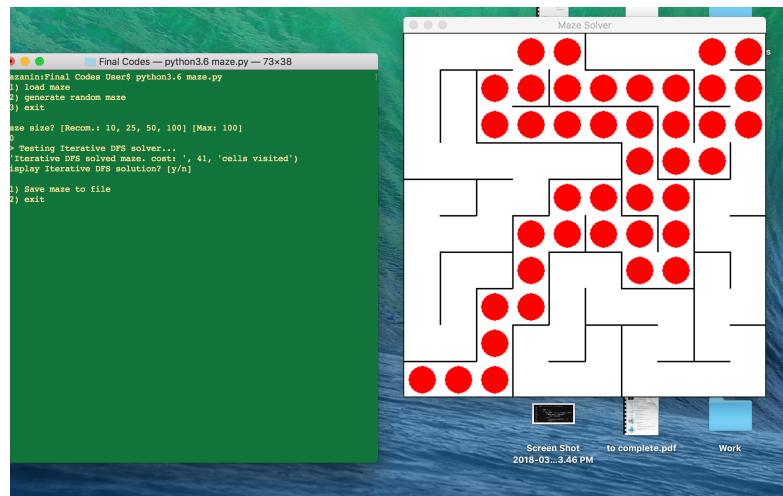
برای نگه داشتن عمق در روش غیر بازگشتی از دیکشنری در پایتون استفاده می‌کنیم به این صورت که عمق ند اول را ۱ می‌گیریم و پس از آن در هر خانه، عمق خانه‌های مجاور را $+1$ خانه‌ای که الان در آن هستیم می‌گیریم و به همین صورت ادامه می‌دهیم اگر به جایی رسیدیم که عمق خانه‌ای که در آن هستیم از عمق مجاز بیشتر شد از تابع خارج می‌شویم.

شکل ۱۲ نشان می‌دهد که در هر مرحله Iterative_DFS تنها تا عمق مجاز جلو رفته ایم و این عمق کم کم زیاد شده است.

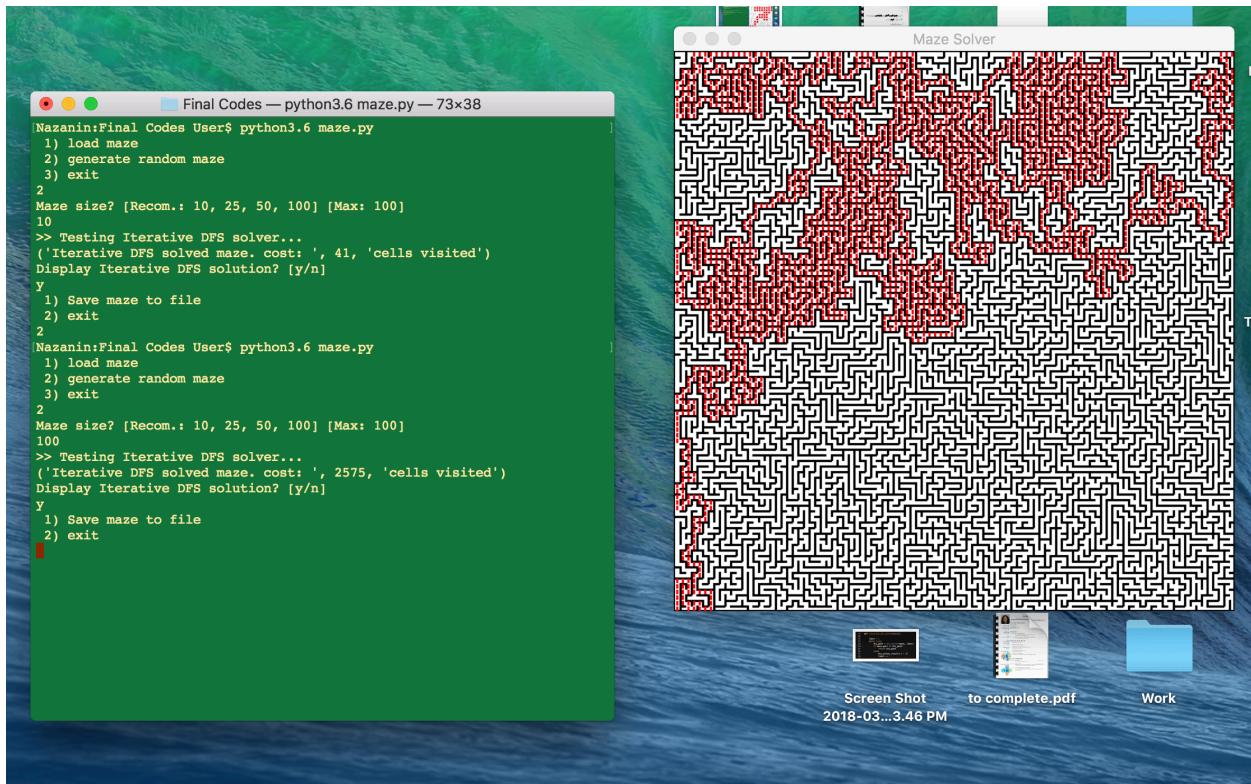
```
>> Testing Iterative DFS solver...
1
[(0, 9)]
2
[(0, 9), (1, 9)]
3
[(0, 9), (1, 9), (2, 9)]
4
[(0, 9), (1, 9), (2, 9), (2, 8)]
5
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7)]
6
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7), (1, 7)]
7
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7), (1, 7), (0, 7)]
8
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7), (1, 7), (0, 7), (0, 6)]
9
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7), (1, 7), (0, 7), (0, 6), (0, 5)]
10
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7), (1, 7), (0, 7), (0, 6), (0, 5),
(0, 4)]
11
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7), (1, 7), (0, 7), (0, 6), (0, 5),
(0, 4), (1, 4)]
12
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7), (1, 7), (0, 7), (0, 6), (0, 5),
(0, 4), (1, 4), (2, 4)]
13
[(0, 9), (1, 9), (2, 9), (2, 8), (2, 7), (1, 7), (0, 7), (0, 6), (0, 5),
(0, 4), (1, 4), (2, 4), (3, 4)]
14
```

شکل ۱۲: زیاد شدن آرام آرام حد مجاز عمق و طولانی تر شدن مسیرهای دیده شده به همین نسبت در Iterative_DFS

شکل‌های ۱۳ و ۱۴، ۲ بار اجرای این کد را به ازای ماز به اندازه‌ی ۱۰ و ۱۰۰ نشان می‌دهند.



شکل ۱۳: اجرای Iterative_DFS برای ماز با اندازه‌ی ۱۰



شکل ۱۴: اجرای Iterative_DFS برای ماز با اندازه‌ی ۱۰۰

Astar_Solver

به طور کلی این الگوریتم به این شکل است عمل می‌کند که در هر مرحله از میان تمام ندهایی که به آن‌ها دسترسی داریم ندی را انتخاب می‌کنیم که با کمترین هزینه می‌تواند به هدف (مقصد) برسد. پس از انتخاب این ند آن را به دسته‌ی ندهای بررسی شده اضافه می‌کند و به سراغ مجاورهای آن می‌رود به این شکل که در صورتی که قبل از بررسی نشده باشند یا در حال بررسی نباشند آن‌ها را به لیست در حال بررسی اضافه می‌کنیم و بعد هزینه‌ی اینکه از این مسیر به این خانه برسیم را محاسبه می‌کنیم اگر این هزینه کمتر از رسیدن از هر خانه‌ی دیگری که تا الان دیده‌ایم به این خانه بود خانه‌ی قبلی این ند را برابر با خانه‌ی در حال بررسی قرار می‌دهیم و هزینه‌ها را نیز به روز رسانی می‌کنیم.

تابع هیوریستیک استفاده شده در این سوال از جنس فاصله است یعنی فاصله‌ی ند در حال بررسی تا هدف. از آنجا که در طول الگوریتم مسیر را به صورت خانه‌ی قبلی هر خانه ذخیره کرده بودیم حال باید از روی این مقادیر مسیر اصلی را بسازیم، به این صورت که می‌گوییم خانه‌ی قبلی هدف X بوده و خانه‌ی قبلی Y بوده است و به همین ترتیب ادامه می‌دهیم تا به مبدا برسیم.

مقایسه

در جدول‌های زیر به مقایسه‌ی این ۴ روش با مازهای یکسان می‌پردازم (عدددها تعداد خانه‌های دیده شده اند):

جدول ۱: مقایسه به ازای ماز به اندازه‌ی ۱۰

BFS	DFS	Iterative DFS	A*
57	41	41	41

جدول ۲: مقایسه به ازای ماز به اندازه‌ی ۵۰

BFS	DFS	Iterative DFS	A*
992	602	602	583

جدول ۳: مقایسه به ازای ماز به اندازه‌ی ۱۰۰

BFS	DFS	Iterative DFS	A*
5892	2773	2773	2475

فایل‌های آپلود شده

۱. گزارش پروژه
۲. فایل‌های اصلی پروژه