

Plot and Navigate a virtual maze

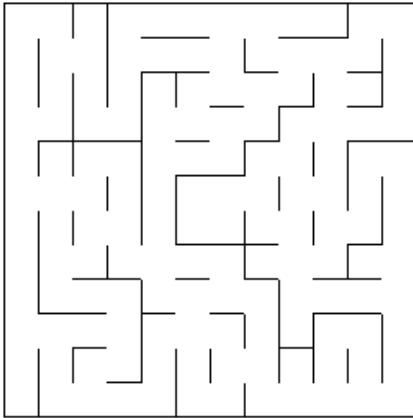
CAPSTONE PROJECT

Prateek Kacker | MLE Nanodegree | December 27, 2016

Definition

PROJECT OVERVIEW

This project takes inspiration from Micromouse competitions in which mouse is tasked with plotting a path through a virtual maze to its center. In the first run, the mouse can explore the virtual maze with little penalty but in second run it must reach the center of virtual maze as quickly as possible. Though the mouse is allowed 1000 moves to reach the center but its objective is to reach the center twice as quickly as possible because a score is summed up for both the trial based on the number of moves and the mouse with the lowest score is better than the mouse with higher score. Typical virtual maze would look like the following: -



PROBLEM STATEMENT

The input to the problem is a virtual maze and it can be of dimensions 12x12, 14x14 or 16x16. The maze is enclosed from all sides with boundaries. Each cell within the maze also has boundaries. The mouse starts from cell (0,0) and facing upwards. The mouse can sense the number of blocks that are available to walk in front, left and right direction. In one move, the mouse can only move up to three moves and in the forward, right and left direction. The mouse can choose to not move at all but it can rotate if it is required. The mouse has two trial runs. At the beginning, the mouse does not have any knowledge of the virtual maze hence it should explore the maze in the first trial run. If it has reached the center, the mouse is given choice to end the trial run and start the second run. In this run, it tries to reach the center as quickly as possible. The maximum number of allowed moves is 1000 for the two runs.

METRICS

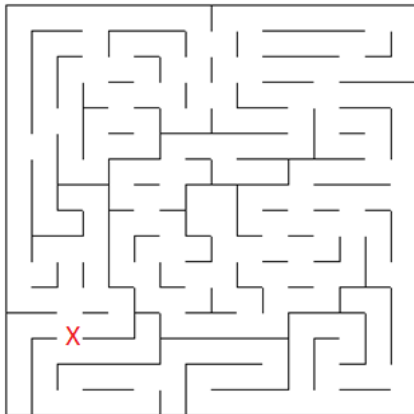
The scores for the each simulation for the mouse is equal to the sum of one thirtieth of the first trial run moves and number of second trial run moves. The aim of this simulation is to reduce number of moves taken to reach the center twice.

Analysis

DATA EXPLORATION

In the first trial run, the mouse explores the virtual maze randomly but giving priority to reaching to the center of the virtual maze because that is the objective of the mouse. The mouse receives signals from three sides i.e. left, front and right. Based on the signals, the mouse creates a maze in its memory by memorizing the barriers and the free movements possible in all four directions. The information is store in *self.Q* dictionary and key of the dictionary is state of the maze or each cell number identified by (x,y) starting from right and bottom corner. The value of the state is dictionary which contains the 0,1,2,3 as keys. 0 denotes if the left direction. If there is 1 as value for the key 0, it would mean that for the cell in the maze, there is no barrier in the left side of the maze. Similarly, 1 denotes the up direction, 2 denote right direction, 3 denote bottom direction. 1 as value to these keys would mean that there is no barrier in that direction. There is another key *utility* which denotes the utility of the cell (or the state) of the maze

In each move, the mouse moves randomly in the direction of the free movement but it gives preference to the direction of the center of the maze because it must reach there before it can start second run. Once the mouse has reached the center of the maze in the first trial run, the mouse explores the maze randomly till it has explored a fixed percentage of the maze. After that it stops doing it and sends a reset signal to the system. The characteristics of maze becomes very important for the mouse so that he can explore it fully. Consider the following maze 3. It has 16 cells square



The mouse prefers going in the direction of center. Considered the cell marked with **x**. At the cell the mouse would prefer to go upward than downwards because of the mouse gives higher priority to move in that direction. If the mouse sets of the journey in the top direction than he might take a longer time to reach to the center because that path can take him in wrong direction. If the mouse moves down which is less likely the mouse will reach the center quickly. These decisions can make or break the mouse trial run in the maze.

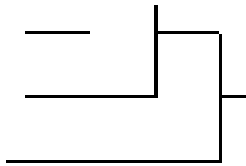
Before the beginning of the second run, the mouse implements Q-learning algorithm for value iteration to find the utility of each state given 500 is the reward of reaching the center of the maze and for each cell movement there is a reward of -1. The value iteration algorithm updates utility of each state to find the best policy to reach the center of the maze. The mouse may not have explore

the maze fully so the algorithm uses the best of the available data to get the best route to the center.

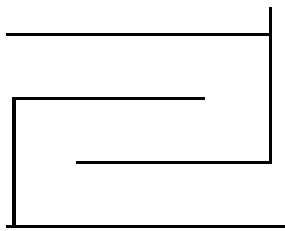
For this maze, the best metric which I have received is 48.

EXPLORATORY VISUALIZATION

In this maze, there are many dead-ends which the mouse might traverse. The mouse might have to travel to the end to realize that he has reached in one. This adds to the number of steps to the trail run. It is not possible for mouse to find out if it is inside them till he goes through them. These dead-ends can be just one cell or many cells.



The best cells are those which do not have many options to decide. In these cells, the mouse will move the most quickly. Here is the example of the cell in which the mouse will move the fastest.



ALGORITHMS AND TECHNIQUES

The mouse uses reinforcement learning algorithm to perform in the second run. The reinforcement learning algorithm works by identifying states in a problem and finding an optimal transition policy between states such that maximum benefits are achieved from the system. The problem in this simulation is reducing number of moves for the mouse. For this problem, it is assumed that every movement penalizes the mouse by 1 unit. Assuming that the reward at the center of maze is 500 units, the mouse has to reduce the number of steps to gets the maximum number of units as rewards. The algorithm implements this by assigning *utility* of the center of the maze to 500 and rest of the blocks to 0. In each cycle and for each cell, the algorithm looks for utility of state (or *next_state*) adjacent to the current *state* and if the utility is greater, it changes the utility of the *current state* to utility of the *next_state* - 1. The algorithm then iterates it to $maze_dimension * maze_dimension$ time so that benefits reach all cells. Intuitively this means that if it takes one step to reach *next_state* then the utility of the current state should be one less than utility of the *next_state*. If we start from center which has 500 utility, we can get to know the utility of correct utility of each next to it and then further in $maze_dimension * maze_dimension$ times.

In order to implement the above algorithm, the mouse has to create transition model (Q-learning model). In the first trial run, the mouse creates that by randomly running around the maze. Before the second run, the mouse calculates the optimal policy of each state based on transition models.

At the beginning, the mouse then starts from cell (0,0) and keeps on moving in the direction which increases its utility. If there is a chance to move in multiple steps because the utility is increasing in the direction, the mouse takes multiple steps in that direction. There are other techniques which mouse use. The first one is weighted random exploration. In the first run, the mouse has upto 3 directions to go based on the free movement available to the mouse. The mouse gives weight to each direction because if it can lead to the center of the maze. The weight vectors are modelled by variables high, medium and low. A random number is generated and based on the ranges from normalized vectors, the mouse will move in that directions. This technique is not perfect but it gives better results than complete random walk in the maze. Second technique is the second trial run. The mouse choose to take upto 3 steps based on the optimum policy. The mouse starts off with 1 step and if there is no other barrier in the direction of movement, he takes two steps or three steps. This reduces number of steps taken to move forward.

BENCHMARK:

In a 16x16 maze, the optimum number of steps will be 30-50 steps or more. It is because it will take around 20-30 steps to reach the center in second trial and then 300-600 steps (equivalent 10-20 steps) to explore the mouse in first trial. If the mouse takes a wrong decision at crucial junctions, then he may not even reach the center of maze in 1000 steps because it travels randomly.

Methodology

DATA PREPROCESSING

The sensors data from the mouse is preprocessed before storing in the memory. The sensors give information about the number of cells which are available to move in front, left and right directions. The mouse only extracts the information that if there is a barrier in these three direction for each cell. It then creates an image of the cell with barrier information in four directions. It has been implemented in the following code.

```
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

IMPLEMENTATION

There is one major algorithm in the implementation. The mouse takes decisions using two techniques which is of interests here.

The algorithm for the implementation is value iteration on utility functions for the Q-learning models. For each state, the algorithm looks for states which is next to the current and does not have a barrier in the middle. For these *next_state*, if the utility is two more than the current state, then the current state's utility is increased to one minus the utility of the *next_state*. This algorithm is run on each state *maze_dim* * *maze_dim* times. There was no complication running the algorithm because there is no trial and error in this algorithm. It is a standard algorithm and works pretty fine.

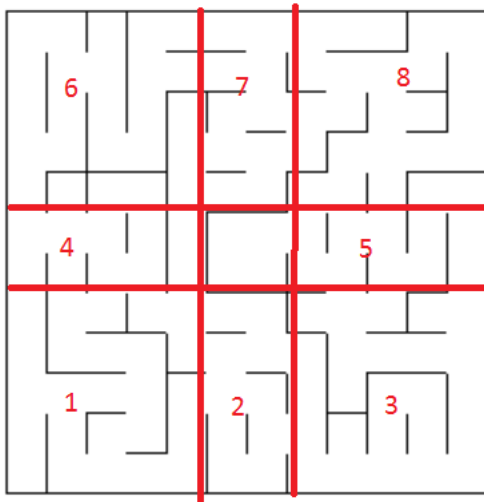
```

413         for z in range(self.maze_dim*self.maze_dim):
414             for i in range(self.maze_dim):
415                 for y in range(self.maze_dim):
416                     state= (i,y)
417                     if state in self.Q:
418                         if (self.Q[state][0] > 0):
419                             state_next=(i-1,y)
420                             if (state_next in self.Q):
421                                 if (self.Q[state]['utility'] <self.Q[state_next]['utility']-1) and self.Q[state_next]
422                                     ['utility']>0:
423                                     self.Q[state]['utility'] =self.Q[state_next]['utility']-1
424                         if (self.Q[state][1] > 0):
425                             state_next=(i,y+1)
426                             if state_next in self.Q:
427                                 if (self.Q[state]['utility'] <self.Q[state_next]['utility']-1) and self.Q[state_next]
428                                     ['utility']>0:
429                                     self.Q[state]['utility'] =self.Q[state_next]['utility']-1
430                         if (self.Q[state][2] > 0):
431                             state_next=(i+1,y)
432                             if state_next in self.Q:
433                                 if (self.Q[state]['utility'] <self.Q[state_next]['utility']-1) and self.Q[state_next]
434                                     ['utility']>0:
435                                     self.Q[state]['utility'] =self.Q[state_next]['utility']-1
436                         if (self.Q[state][3] > 0):
437                             state_next=(i,y-1)
438                             if state_next in self.Q:
439                                 if (self.Q[state]['utility'] <self.Q[state_next]['utility']-1) and self.Q[state_next]
440                                     ['utility']>0:
441                                     self.Q[state]['utility'] =self.Q[state_next]['utility']-1

```

The first process happens in the first trial run. In that run, the mouse takes decision about the direction to move. It is based on the weights of the direction and probability generated by random function. The weights are higher to move it in towards the center of the maze. Hence the maze is divided into 8 blocks and each block have preference directions towards the center of the maze. Based on the output of random function, the mouse moves and it is more probable towards the center.

The blocks are divided as shown below.



In block 1, moving up and right is preferred over moving down and moving right. The weights are given by *high* and *low*. The same kind of interpretation is applied to blocks 3, 6 and 8. In block 2, the direction up is *high*, the direction left and right is *medium* and the down is *low*. Same kind of logic can be applied to 2, 4, 5, and 7.

The code for the weight vectors is show below.


```

action_taken=0
high = 1.2
medium = 1.1
low = 1
weight_vector_block_1=dict()
weight_vector_block_1[0]={0:low, 1:low, 2:high, 3:high}
weight_vector_block_1[1]={0:low, 1:high, 2:high, 3:low}
weight_vector_block_1[2]={0:high, 1:high, 2:low, 3:low}
weight_vector_block_1[3]={0:high, 1:low, 2:low, 3:high}
weight_vector_block_2 = dict()
weight_vector_block_2[0]={0:low, 1:medium, 2:high, 3:medium}
weight_vector_block_2[1]={0:medium, 1:high, 2:medium, 3:low}
weight_vector_block_2[2]={0:high, 1:medium, 2:low, 3:medium}
weight_vector_block_2[3]={0:medium, 1:low, 2:medium, 3:high}
weight_vector_block_3 = dict()
weight_vector_block_3[0]={0:low, 1:high, 2:high, 3:low}
weight_vector_block_3[1]={0:high, 1:high, 2:low, 3:low}
weight_vector_block_3[2]={0:high, 1:low, 2:low, 3:high}
weight_vector_block_3[3]={0:low, 1:low, 2:high, 3:high}
weight_vector_block_4 = dict()
weight_vector_block_4[0]={0:medium, 1:low, 2:medium, 3:high}
weight_vector_block_4[1]={0:low, 1:medium, 2:high, 3:medium}
weight_vector_block_4[2]={0:medium, 1:high, 2:medium, 3:low}
weight_vector_block_4[3]={0:high, 1:medium, 2:low, 3:medium}
weight_vector_block_5 = dict()
weight_vector_block_5[0]={0:medium, 1:high, 2:medium, 3:low}
weight_vector_block_5[1]={0:high, 1:medium, 2:low, 3:medium}
weight_vector_block_5[2]={0:medium, 1:low, 2:medium, 3:high}
weight_vector_block_5[3]={0:low, 1:medium, 2:high, 3:medium}
weight_vector_block_6 = dict()
weight_vector_block_6[0]={0:high, 1:low, 2:low, 3:high}
weight_vector_block_6[1]={0:low, 1:low, 2:high, 3:high}
weight_vector_block_6[2]={0:low, 1:high, 2:high, 3:low}
weight_vector_block_6[3]={0:high, 1:high, 2:low, 3:low}
weight_vector_block_7 = dict()
weight_vector_block_7[0]={0:high, 1:medium, 2:low, 3:medium}
weight_vector_block_7[1]={0:medium, 1:low, 2:medium, 3:high}
weight_vector_block_7[2]={0:low, 1:medium, 2:high, 3:medium}
weight_vector_block_7[3]={0:medium, 1:high, 2:medium, 3:low}
weight_vector_block_8 = dict()
weight_vector_block_8[0]={0:high, 1:high, 2:low, 3:low}
weight_vector_block_8[1]={0:high, 1:low, 2:low, 3:high}
weight_vector_block_8[2]={0:low, 1:low, 2:high, 3:high}
weight_vector_block_8[3]={0:low, 1:high, 2:high, 3:low}

```

In the second run, to optimize the number of moves, the mouse should move in multiple steps if the maze allows and takes you towards the center of the maze. The mouse first finds the right policy on the cell which he is standing. He then checks for cells in the direction of its movement to find out if he should take 1, 2, 3 steps and improve the utility of the state.


```

97         else:
98             if (self.location[0]-2,self.location[1]) in self.Q and self.Q[self.location[0]-1,self.location[1]][max_var]
99             >0:
100                 if self.Q[self.location[0]-2,self.location[1]][max_var]>0 and self.Q[self.location[0]-2,self.location[1]]
101                 ['utility'] > max_utility:
102                     max_utility_2= self.Q[self.location[0]-2,self.location[1]]['utility']
103                     if (self.location[0]-3,self.location[1]) in self.Q:
104                         if self.Q[self.location[0]-3,self.location[1]][max_var]>0 and self.Q[self.location[0]
105                         -3,self.location[1]][max_var] > max_utility_2:
106                             self.location[0]=self.location[0]-3
107                             movement=3
108                         else:
109                             self.location[0]=self.location[0]-2
110                             movement=2
111                         else:
112                             self.location[0]=self.location[0]-1
113                             movement=1
114                     else:
115                         self.location[0]=self.location[0]-1
116                         movement=1
117             elif max_var==1:
118                 if self.heading == 3:
119                     movement = 0
120                 else:
121                     if (self.location[0],self.location[1]+2) in self.Q and self.Q[self.location[0],self.location[1]+1][max_var]

```

REFINEMENT

There was no refinement of algorithm required because the algorithm is a standard reinforcement algorithm.

Results

MODEL EVALUATION AND VALIDATION

The mouse simulation is perfect to understand the power of value iteration algorithm in Q-Learning. The mouse develops a model of the maze and then performs the value iteration to find the best policy to reach the center of the maze. The value iteration guarantees the shortest possible time to reach the center of the maze. The second trial run is very important because every step adds directly the final score. The first run is where the mouse can use different strategies to explore the maze. There can be many different strategies for exploring and they may have different impact on the result.

The following results have been obtained till now with weights *high*, *medium* and *low* as 1.2, 1.1 and 1.0 respectively and percentage covered as 70%

Runs	12x12	14x14	16x16
1	33.267	34.867	41.467
2	28.8	52.467	Unable to complete
3	23.67	56.733	45.43
4	23.73	48.767	48.3
5	26.87	Unable to complete	44.8
6	33.167	51.7	40.13
7	36.33	56.267	40.667
8	29.63	50.267	45.833
9	Unable to complete	47.33	50.2
10	28.0	45.233	51.70

Performance of the 12x12 for different *high*, *low* and *medium* configurations are as follows:-

<i>high, medium, low</i>	(1,1,1)	(1.1,1.05,1.0)	(1.2,1.1,1.0)	(1.4,1.2,1.0)
1	41	32.067	30.833	Unable to complete
2	Unable to complete	Unable to complete	35.367	42.167
3	24.233	29.733	23.467	Unable to complete
4	28.0	28.833	29.20	41.867
5	29.6	26.80	23.30	51.80

Clearly the best performance for the mouse is when the *high*, *medium* and *low* are 1.2,1,1 and 1.0.

Performance of the 12x12 for different *percentage_covered* configurations are as follows:-

Percentage	60%	70%	80%	90%
1	33.40	30.267	35.50	Unable to complete
2	40.333	22.233	Unable to complete	34.433
3	38.633	21.90	29.867	37.60
4	26.667	30.50	28.10	37.90
5	24.70	22.533	32.10	32.367

We see the same effect here. As the percentage goes up, chances that mouse is unable to finish exploring higher percentage of maze increases and it decreases the performance.

The mouse requires working on two parameters for better performance. These parameters are incorporated in the techniques discussed above. The first parameter is *percentage_covered*. This variable decides how much the mouse explores the maze in first trial in percentage before starting second trial. This is very important factor because this factor can drastically increase the number of trials for the mouse if it is very high number. The optimum value is around 70% for this simulation. This factor is very important for the simulation because if it was 100% then we would never be able to complete any challenge. If is less than we will get less than optimum results. With lower values around (50-60%), I have seen the metric taking around 80 to 90 units to finish.

The second parameters are the weights for the direction which the mouse uses to prioritize his movement towards the center of maze. Those are three variables *high*, *medium* and *low* and it gets normalized hence the relative ratios are important to get absolutely probability range. Currently I have assigned the variables *high*, *medium* and *low* to 1.2, 1.1 and 1.0 respectively. The performance with *high*, *medium* and *low* is average 80 when all the values are set to 1. The performance goes down when the weights are increased because there are several crucial cells where the better decision is different from weights of the directions. Wrong decision in those cells have increased the trial time. In some cases, the mouse has not been able to complete the maze at all.

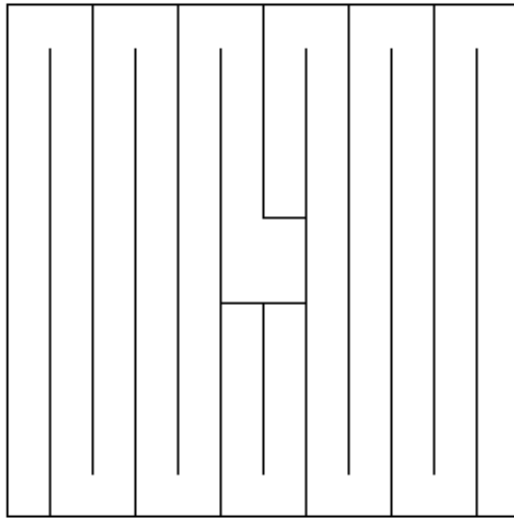
JUSTIFICATIONS

The results justify the model that it is good for the simulation. The results obtained for 16x16 average around 45 units. Based on rough calculations, the optimum number of steps for the maze is around 25-30 steps and the maze requires around 400-600 steps in stage 1. This adds up to the approximately 45 units. There are several things which can be improved in this model but they will add to complexity of the algorithm and may be very specific to one of the problems in the maze. This way the algorithm will overfit to the testing maze.

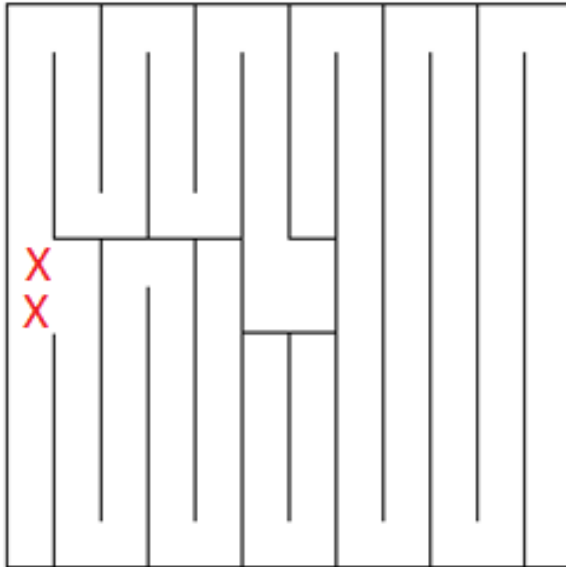
Conclusion

FREE FORM VISUALIZATION

Maze architecture can improve or reduce the performance of the mouse. Consider the following maze. This kind of the maze is easiest to solve because the mouse has only one option to go which is the forward direction. The maze guides the mouse center of the maze. The weight vectors do not provide any guidance because the mouse has only one option which is to move forward.

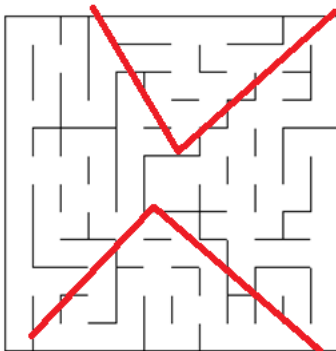


Consider the following maze shown below. There are two places where the weights of direction would increase the chances of the mouse going towards wrong direction and hence wasting the important trial runs.



Most of the time, the maze will have the architecture which will get benefitted from the weights.

Consider the following maze. In this maze, if the mouse is on the right or the left side then any movement towards the center will help the mouse to reach the center as soon as possible.



REFLECTION

The challenge starts with mouse exploring the maze in the first trial. The mouse starts from cell (0,0) and facing upwards. The mouse captures information from the sensors, processes and stores in *self.Q* dictionary with cell number as state name.

```

194 -----
195 if self.learning == True: ##### Trial run 1
196     location_tuple = (self.location[0],self.location[1]) ##### initialization function
197
198     ##### This section helps you to pre-process information from the sensors and convert it to barrier information
199
200     if sensors[0]>0:
201         anticlockwise=1
202     else:
203         anticlockwise=0
204     if sensors[1]>0:
205         straight = 1
206     else:
207         straight = 0
208     if sensors[2]>0:
209         clockwise=1
210     else:
211         clockwise=0
212
213     ##### Based on the direction of the movement, the sensor information should be rotated to get the exact map of the
214     maze
215     if (location_tuple in self.Q):
216         a=1
217     else:
218         if self.heading == 1:
219             self.Q[location_tuple] = (0:anticlockwise, 1:straight, 2:clockwise, 3:1)
220         elif self.heading == 2:
221             self.Q[location_tuple] = (0:1, 1:anticlockwise, 2:straight, 3:clockwise)
222         elif self.heading == 3:
223             self.Q[location_tuple] = (0:clockwise, 1:1, 2:anticlockwise, 3:straight)
224         else:
225             self.Q[location_tuple] = (0:straight, 1:clockwise, 2:1, 3:anticlockwise)

```

In order to move ahead the mouse chooses his direction based on random probability but giving weight to directions based on the philosophy that it has reach the center as quickly as possible.

```

225 high = 1.2 ## Initializing probability variables
226 medium = 1.1 ##
227 low = 1
228 ## Creating a weight vector blocks for each quadrant
229 weight_vector_block_1=dict()
230 weight_vector_block_1[0]=(0:low, 1:low, 2:high, 3:high)
231 weight_vector_block_1[1]=(0:low, 1:high, 2:high, 3:low)
232 weight_vector_block_1[2]=(0:high, 1:high, 2:low, 3:low)
233 weight_vector_block_1[3]=(0:high, 1:low, 2:low, 3:high)
234 weight_vector_block_2 = dict()
235 weight_vector_block_2[0]=(0:low, 1:medium, 2:high, 3:medium)
236 weight_vector_block_2[1]=(0:medium, 1:high, 2:medium, 3:low)
237 weight_vector_block_2[2]=(0:high, 1:medium, 2:low, 3:medium)
238 weight_vector_block_2[3]=(0:medium, 1:low, 2:medium, 3:high)
239 weight_vector_block_3 = dict()
240 weight_vector_block_3[0]=(0:low, 1:high, 2:high, 3:low)
241 weight_vector_block_3[1]=(0:high, 1:high, 2:low, 3:low)
242 weight_vector_block_3[2]=(0:high, 1:low, 2:low, 3:high)
243 weight_vector_block_3[3]=(0:low, 1:low, 2:high, 3:high)
244 weight_vector_block_4 = dict()
245 weight_vector_block_4[0]=(0:medium, 1:low, 2:medium, 3:high)
246 weight_vector_block_4[1]=(0:low, 1:medium, 2:high, 3:medium)
247 weight_vector_block_4[2]=(0:medium, 1:high, 2:medium, 3:low)
248 weight_vector_block_4[3]=(0:high, 1:medium, 2:low, 3:medium)
249 weight_vector_block_5 = dict()
250 weight_vector_block_5[0]=(0:medium, 1:high, 2:medium, 3:low)
251 weight_vector_block_5[1]=(0:high, 1:medium, 2:low, 3:medium)
252 weight_vector_block_5[2]=(0:medium, 1:low, 2:medium, 3:high)
253 weight_vector_block_5[3]=(0:low, 1:medium, 2:high, 3:medium)
254 weight_vector_block_6 = dict()
255 weight_vector_block_6[0]=(0:high, 1:low, 2:low, 3:high)
256 weight_vector_block_6[1]=(0:low, 1:low, 2:high, 3:high)
257 weight_vector_block_6[2]=(0:low, 1:high, 2:high, 3:low)
258 weight_vector_block_6[3]=(0:high, 1:high, 2:low, 3:low)
259 weight_vector_block_7 = dict()
260 weight_vector_block_7[0]=(0:high, 1:medium, 2:low, 3:medium)
261 weight_vector_block_7[1]=(0:medium, 1:low, 2:medium, 3:high)
262 weight_vector_block_7[2]=(0:low, 1:medium, 2:high, 3:medium)
263 weight_vector_block_7[3]=(0:medium, 1:high, 2:medium, 3:low)
264 weight_vector_block_8 = dict()
265 weight_vector_block_8[0]=(0:high, 1:high, 2:low, 3:low)
266 weight_vector_block_8[1]=(0:high, 1:low, 2:low, 3:high)
267 weight_vector_block_8[2]=(0:low, 1:low, 2:high, 3:high)
268 weight_vector_block_8[3]=(0:low, 1:high, 2:high, 3:low)
269

```

```

274         if sensors[0] == 0 and sensors[1] == 0 and sensors[2] == 0:
275             rotation = 90
276             movement = 0
277         else:
278             while action_taken == 0:
279                 if reached_goal == 0:
280                     if (self.location[0] < self.maze_dim/2-1) and self.location[1] < self.maze_dim/2-1:
281                         weight_vectors = weight_vector_block_1[self.heading]
282                     elif (self.location[0] < self.maze_dim/2+1) and (self.location[0] >= self.maze_dim/2-1) and self.location[1] < self.maze_dim/2-1:
283                         weight_vectors = weight_vector_block_2[self.heading]
284                     elif (self.location[0] >= self.maze_dim/2+1) and self.location[1] < self.maze_dim/2-1:
285                         weight_vectors = weight_vector_block_3[self.heading]
286                     elif (self.location[0] < self.maze_dim/2-1) and self.location[1] >= self.maze_dim/2+1 and self.location[1] < self.maze_dim/2-1:
287                         weight_vectors = weight_vector_block_4[self.heading]
288                     elif (self.location[0] >= self.maze_dim/2+1) and self.location[1] >= self.maze_dim/2-1 and self.location[1] < self.maze_dim/2+1:
289                         weight_vectors = weight_vector_block_5[self.heading]
290                     elif (self.location[0] < self.maze_dim/2-1) and self.location[1] >= self.maze_dim/2+1:
291                         weight_vectors = weight_vector_block_6[self.heading]
292                     elif (self.location[0] < self.maze_dim/2+1) and (self.location[0] >= self.maze_dim/2-1) and self.location[1] >= self.maze_dim/2+1:
293                         weight_vectors = weight_vector_block_7[self.heading]
294                     elif (self.location[0] >= self.maze_dim/2+1) and self.location[1] >= self.maze_dim/2+1:
295                         weight_vectors = weight_vector_block_8[self.heading]

296         ##### Normalizaing the weight vectors
297         weight_vector_normalized = {0:float(weight_vectors[0])/(weight_vectors[0]+weight_vectors[1]
+weight_vectors[2]), 1:float(weight_vectors[1])/(weight_vectors[0]+weight_vectors[1]+weight_vectors[2]), 2:float(weight_vectors
[2])/(weight_vectors[0]+weight_vectors[1]+weight_vectors[2])}
298         ##### Deciding on action based on randomly generated probability and ignoring 180 movements
299         prob = random.random()
300         if prob < weight_vector_normalized[0]:
301             action = 0
302         elif prob < weight_vector_normalized[1]+weight_vector_normalized[0]:
303             action = 1
304         else:
305             action = 2

```

If the mouse reached the center then mouse is given an option to explore the maze randomly.

```

306         ##### If the mouse has reached the center once then allowing the mouse to explore the maze more
307         randomly
308         else:
309             action = random.randrange(0,3)
310             if action == 3:
311                 rotation = 90
312                 movement = 0
313                 action_taken = 1
314             elif sensors[action] > 0:
315                 if action == 0:
316                     rotation = -90
317                     movement = 1
318                 elif action == 1:
319                     rotation = 0
320                     movement = 1
321                 else:
322                     rotation = 90
323                     movement = 1
324             action_taken = 1

```

If the mouse reaches the center and explores a fixed percentage of the maze then it sends a reset signal.

```

376         if self.learning_counter == 1000 or (percentage_covered > 70 and reached_goal == 1):
377             self.learning = False
378             self.heading = 1
379             self.location = [0,0]
380             if reached_goal == 1:
381                 rotation = 'Reset'
382                 movement = 'Reset'

```

After ending the first trial the mouse performs the value iteration algorithm.


```

412 ##### Value iteration algorithm in execution
413 for z in range(self.maze_dim*self.maze_dim):
414     for i in range(self.maze_dim):
415         for y in range(self.maze_dim):
416             state= (i,y)
417             if state in self.Q:
418                 if (self.Q[state][0] > 0):
419                     state_next=(i-1,y)
420                     if (state_next in self.Q):
421                         if (self.Q[state]['utility'] < self.Q[state_next]['utility']-1) and self.Q[state_next]
['utility']>0:
422                             self.Q[state]['utility'] =self.Q[state_next]['utility']-1
423                 if (self.Q[state][1] > 0):
424                     state_next=(i,y+1)
425                     if (state_next in self.Q):
426                         if (self.Q[state]['utility'] < self.Q[state_next]['utility']-1) and self.Q[state_next]
['utility']>0:
427                             self.Q[state]['utility'] =self.Q[state_next]['utility']-1
428                 if (self.Q[state][2] > 0):
429                     state_next=(i+1,y)
430                     if (state_next in self.Q):
431                         if (self.Q[state]['utility'] < self.Q[state_next]['utility']-1) and self.Q[state_next]
['utility']>0:
432                             self.Q[state]['utility'] =self.Q[state_next]['utility']-1
433                 if (self.Q[state][3] > 0):
434                     state_next=(i,y-1)
435                     if (state_next in self.Q):
436                         if (self.Q[state]['utility'] < self.Q[state_next]['utility']-1) and self.Q[state_next]
['utility']>0:
437                             self.Q[state]['utility'] =self.Q[state_next]['utility']-1
438
439

```

The output of the algorithm is optimum policy for the mouse to travel to the center in the second trial run. The mouse not only decides the direction but also the number of steps to move in that direction.

```

89 ##### This section is helps to identify the direction of the movement and the number of steps which should be
taken.
90 for i in range(4):
91     if max_utility < utility[i]:
92         max_utility = utility[i]
93         max_var = i
94     rotation=rotation_array[self.heading][max_var]
95     self.heading=self_heading[self.heading][max_var]
96     if max_var==0:##### if the direction decided is going left
97         if self.heading == 2: ##### But we are pointing in right
98             movement = 0
99         else:
100             if (self.location[0]-2,self.location[1]) in self.Q : ##### if state next to next_state in the direction of
max_var is available.
101                 if self.Q[self.location[0]-2,self.location[1]]['utility'] > max_utility and self.Q[self.location[0]
-1,self.location[1]][max_var]>0: ### If that state is accesible and has utility requirements
102                     max_utility_2= self.Q[self.location[0]-2,self.location[1]]['utility']
103                     if (self.location[0]-3,self.location[1]) in self.Q: ##### If the state is next to next to the
next_state direction of max_var.
104                         if self.Q[self.location[0]-2,self.location[1]][max_var]>0 and self.Q[self.location[0]
-3,self.location[1]]['utility'] > max_utility_2: ## If the state is accesible and has utility requirements
105                             self.location[0]=self.location[0]-3
106                             movement=3
107                         else:
108                             self.location[0]=self.location[0]-2
109                             movement=2
110                     else:
111                         self.location[0]=self.location[0]-2
112                         movement=2
113                 else:
114                     self.location[0]=self.location[0]-1
115                     movement=1
116             else:
117                 self.location[0]=self.location[0]-1
118                 movement=1
119

```

```

120         elif max_var==1: #### If the direction decided is going up
121             if self.heading == 3: ##### If point towards down
122                 movement = 0
123             else:
124                 if (self.location[0],self.location[1]+2) in self.Q :
125                     if self.Q[self.location[0],self.location[1]+2]['utility'] > max_utility and self.Q[self.location
126 [0],self.location[1]+1][max_var]>0:
127                         max_utility_2= self.Q[self.location[0],self.location[1]+2]['utility']
128                         if (self.location[0],self.location[1]+3) in self.Q:
129                             if self.Q[self.location[0],self.location[1]+2][max_var]>0 and self.Q[self.location
130 [0],self.location[1]+3]['utility'] > max_utility_2:
131                                 self.location[1]=self.location[1]+3
132                                 movement=3
133                             else:
134                                 self.location[1]=self.location[1]+2
135                                 movement=2
136                         else:
137                             self.location[1]=self.location[1]+2
138                             movement=2
139                     else:
140                         self.location[1]=self.location[1]+1
141                         movement=1
142                 else:
143                     self.location[1]=self.location[1]+1
144                     movement=1
145
146     elif max_var==2: ##### if the direction decided is going right
147         if self.heading==0:
148             movement = 0 ##### if the mouse point towards left
149         else:
150             if (self.location[0]+2,self.location[1]) in self.Q :
151                 if self.Q[self.location[0]+2,self.location[1]]['utility'] > max_utility and self.Q[self.location[0]
152 +1,self.location[1]][max_var]>0:
153                     max_utility_2= self.Q[self.location[0]+2,self.location[1]]['utility']
154                     if (self.location[0]+3,self.location[1]) in self.Q:
155                         if self.Q[self.location[0]+2,self.location[1]][max_var]>0 and self.Q[self.location[0]
156 +3,self.location[1]]['utility'] > max_utility_2:
157                             self.location[0]=self.location[0]+3
158                             movement=3
159                         else:
160                             self.location[0]=self.location[0]+2
161                             movement=2
162                     else:
163                         self.location[0]=self.location[0]+2
164                         movement=2
165                     else:
166                         self.location[0]=self.location[0]+1
167                         movement=1
168                 else:
169                     self.location[0]=self.location[0]+1
170                     movement=1
171             else:
172                 self.location[0]=self.location[0]+1
173                 movement=1
174
175     ---
176     else: ##### if the direction decided is down
177         if self.heading == 1: ##### if the mouse is pointing towards up
178             movement = 0
179         else:
180             if (self.location[0],self.location[1]-2) in self.Q :
181                 if self.Q[self.location[0],self.location[1]-2]['utility'] > max_utility and self.Q[self.location
182 [0],self.location[1]-1][max_var]>0:
183                     max_utility_2= self.Q[self.location[0],self.location[1]-2]['utility']
184                     if (self.location[0],self.location[1]-3) in self.Q:
185                         if self.Q[self.location[0],self.location[1]-2][max_var]>0 and self.Q[self.location
186 [0],self.location[1]-3]['utility'] > max_utility_2:
187                             self.location[1]=self.location[1]-3
188                             movement=3
189                         else:
190                             self.location[1]=self.location[1]-2
191                             movement=2
192                     else:
193                         self.location[1]=self.location[1]-2
194                         movement=2
195                     else:
196                         self.location[1]=self.location[1]-1
197                         movement=1
198                 else:
199                     self.location[1]=self.location[1]-1
200                     movement=1
201             else:
202                 self.location[1]=self.location[1]-1
203                 movement=1
204         ---

```

The most difficult part of the challenge was the value iteration algorithm implementation. Implementing it correctly was tricky since there are lot of conditions associated with the algorithm which should be taken care of while implementing the algorithm. One of the conditions is that the cell adjacent to the current cell should have access to the cell and there should not be a wall between the cell. All these conditions should be met before utility can be updated.

Improvement

This algorithm can be improved too. Now the value iteration algorithm is done choosing the *next_state* as one adjacent state or cell. However, the mouse can jump 3 steps at time hence algorithm should be modified to achieve that. The current implementation tries to do the same thing but in the second trial run when the mouse decides to take a single step or multiple steps. This can be implemented by changing the value iteration algorithm by not only analyzing the adjacent cell for the *utility* value but also the cells which are 2 or 3 steps away from the current cell. This way 2-3 step information will get encoded in *utility* value of each cell.

If the maze was setup in continuous domain. For example, each square has a unit length, walls are 0.1 units thick, and the robot is a circle of diameter 0.4 units. The algorithm will modify slightly to accommodate the dimensions of the maze. Taking the minimum size of 0.1 units, each of the dimensions will have number of unit assigned to it. A single cell will be 10 units and mouse is 4 units a wall is 1 unit. The algorithm must keep track of the units travelled to accommodate the movement of mouse across the maze.