1 Introduction

1.1 Problem overview

The task is to simulate in prolog using 3 different algorithms a regby game round with the following rules:

- Human with a ball can move 1 step in 4 directions: up/down/left/right with cost 1.
- Human can't step on a cell with an ork
- Human can see 1 cell in each of moving directions
- Once per round human can give a pass with cost 1 in 4 basic directions + 4 diagonals if there exists human to get the pass and no ork standing on ball's path.
- If a human steps on a cell with another human, the ball is transferred and human's last move is free.
- If human steps on a touchdown point, the round is over.

2 Implemented algorithms

Note: refer section 3 to see how to run a particular algorithm

2.1 Backtracking

The essence of the backtracking technique is to find the best solution (the shortest path to a touchdown in our case) by traversing the whole search tree. If action results in a prohibited state, the algorithm stops exploration of a branch reverting last action and proceed evaluation other actions. Possible way of implementing the algorithm is to find all available paths and choose the shortest one. The algorithm uses prolog tree to try all possible paths. In order for algorithm not to loop, I prohibit it to go to cell, which it has already passed. The recursion is as follows:

$$backtrack(pos) = \begin{cases} last\_step\_to\_touchdown, \text{if the touchdown is within 1 step from current position} \\ step + backtrack(new\_pos), \text{otherwise} \end{cases}$$

The table 1 briefly describes all predicates related to the backtracking search

find_best_path/2	Get the shortest path via backtrackig.
select_element/2	Choose the best element from a list based on function.
get_better_path/3	Return the shorterst path
find_touchdown/3	Find some Path and its score (param. Optimized stands for algorithm)
go/8	Find next stete from Xstart, Ystart, PathStart, CurrentScore, PassedThisRound

Table 1: Explaining functions involved in backtracking search

2.2 Random search

For a random search there were an ambiguity in task description whether we should implement a search that will run search once with a limit of 100 steps or search 100 times without a limit. Thus I have implemented an algorithm that accepts step limit and how many times it should run the search.

The choice of action on every step is a random choice between all valid and invalid actions, however I decided to make a little improvement: if an agent see the border of the playground it will not throw the ball out of bounds or step out of the field. Such approach helped to increase convergence rate of an algorithm, so that on the first step with a chance $\frac{3}{5}$ the agent will throw a ball somewhere and with a chance $\frac{2}{5}$ make a step instead of $\frac{8}{12}$ and $\frac{4}{12}$ correspondingly including actions resulting in going to negative coordinates.

Usually random search fails, since probability to throw a ball out of the map is $\frac{8}{12}$ if standing in the middle of an empty field, but there are some maps, when it finds very good results. More on this in section 4.

Table 2 demonstrates brief explanation of predicates involved in random searching

run_random/4	Get best answer from random with #runs and step limit
run_random_/4	A utility function to run search 'Times' times
random_search/3	Make 1 random run with a step limit
generate_random_path/8	Make a random path from state
make_random_step/8	Choose 1 step at random from passed state
get_possible_steps/6	Returns an array of all possible steps from given state

Table 2: A brief explanation of predicates used for random search

2.3 Optimized Backtracking

As a 3^{rd} algorithm I have decided to introduce a little trick for a backtracking that drastically improves the performance. After finding some path via regular backtracking of length n, the algorithm will cut-off all branches with length exceeding n. When the better path is found, it is saved. Despite an overhead of comparing number of steps after each action, overall computational complexity is reduced due to cutting branches earlier. The main problem is the fact that the algorithm highly depends on the length of first found paths. Faster it finds short path, less computations it will have to performs to find another path

The predicates involved in the optimized backtracking search are essentially the same as for usual backtracking(table 1), but with a parameter $optimized = true$.

3 Running instructions

3.1 Requirements

- python3
- swipl

3.2 Preparation for execution

3.2.1 According to an assignment

0. Open an assignment folder.
1. Paste the map in form of predicates into the file `input.pl` .
2. In `regby.pl` uncomment the predicate `fieldSize` on line `14` and set the proper size.
3. Run `swipl regby.pl`

3.2.2 Easy way to run maps

0. Open an assignment folder.
1. Draw the map in file `map.txt`. Following the rules :
   - T stands for a touchdown
   - O stands for an ork
   - H stands for a human
   - Bottom left corner is (0, 0)
   - There is a player with ball at (0, 0) by default
2. Make sure the line 14 containing predicate `fieldSize` in `regby.pl` is commented
3. Run `python3 field_generator.py && swipl regby.pl`

3.3 Execution instructions

After running swipl you can run one of execution predicates which executes an algorithm and outputs prettified results with execution time.

- `out_backtracking` will execute simple backtracking search.
- `out_random` will execute random search (100 runs by no more than 100 steps)
- `out_optimized` will execute optimized backtracking search.

4 Testing and Statistics

4.1 Corner cases

There exists several corner cases in which different algorithms behaves differently (with respect to execution speed), but eventually come to expected result (finding the best path for backtracking algorithms and finding or not finding some path). Table 3 shows execution time

and path length found by each algorithm. Visualized test cases can be found at *tests/corner* directory

1. Small path count tests – `maze_small, maze, spiral, rand_2, loop`

   These tests are aim for backtracking algorithms since representing a maze, have quite small number of different paths. As we see, both backtracking algorithms give a precise result, but the bigger map gets, the more time backtracking algorithm require.

   Since the possibility to hit an ork in the maze is very high, random search didn't find any answer.

2. Throw-oriented tests – `diagonal_throw`

   These test require to make a throw in order to reach the best path. All algorithms successfully pass the test, since every algorithm prefer making a throw over making a move.

3. Sparse tests – `sparse_num`

   An empty map with dimensions $num * num$ and a touch down in top right corner.

   As we can see, Optimized backtrack works $\approx$ 10 times faster than the original one. Random algorithm doesn't find any suitable answer due to tendency to throw the ball out of bounds.

4. Pass-catchers – `rand_2, path_of_humans, humans_everywhere, rand_3`

   These type of maps when each pass can be caught by a human. Despite such maps are very rare, they have one interesting property.

   As we can see, all such tests are successfully passed by a random search equally good or even better than optimized backtracking. An interesting case is the test `rand_3` which was successfully passed by a random with a result of 13 in 0.38 sec which is 2 times worse score than optimized backtracking but 15 times faster.

5. Utility tests – `imp_initial_ork, initial_touchdown, imp_blocked_T, humans_along_line, multiple_T, imp_no_T`

   These group of tests is designed to test a response of every algorithm to unusual conditions: no path, no touchdown, ork at (0;0), etc...

   As we can see, all algorithms successfully passed.

As a result we can describe what maps are easy and hard for a particular algorithms:

- **Backtracking**
  W. r. t execution time
  - Easy: fields with small amount of possible moves

  - Hard: sparse maps with a lot of possible moves

- **Random search** W. r. t convergence
  - Easy: maps, where pass is almost always guaranteed to be caught, very small maps.
  - Hard: sparse maps, maze-like maps

- **Optimized backtracking** W. r. t execution time
  - Easy: maps, where one of shortest paths is found first by a regular backtracking.
  - Hard: maps with a lot of equally long paths.

Test name	Backtracking		Random		Optimized Backtracking	
	Score	Time (sec.)	Score	Time (sec.)	Score	Time (sec.)
maze_small	20	0.1	No	0.17	20	0.13
maze	25	1.8	No	0.07	25	0.38
diagonal_throw	3	0.07	3	0.15	3	0.05
sparse_5	8	3.32	No	0.30	8	0.53
sparse_6	10	66.12	No	0.44	10	1.12
sparse_7	12	585.00	No	0.35	12	6.79
spiral	170	0.22	No	0.30	170	0.32
rand_2	1	0.23	1	0.40	1	0.29
pass_only	?	DNF (>7 hours)	No	0.21	15	5.34
path_of_humans	1	0.25	2	0.23	1	0.16
imp_initial_ork	No	0.22	No	0.42	No	0.29
initial_touchdown	0	0.21	0	0.39	0	0.27
loop	8	0.09	No	0.22	8	0.11
imp_blocked_T	No	0.13	No	0.31	No	0.20
humans_everywhere	2	21.75	2	0.35	2	0.25
rand_3	?	DNF	13	0.38	7	6.07
humans_along_line	6	1.10	No	0.16	6	0.14
multiple_T	4	0.14	No	0.31	4	0.19
imp_no_T	No	0.14	No	0.32	No	0.21

Table 3: Speed and result comparison on different tests

4.2 Statistical Analysis

This section presents results of application Analysis of Variance (ANOVA).

Arrangement		SSQ_b	SSQ_w	df_b	df_w	F-Value	P-value
Backtracking vs random	time	2564.1632	39848.5473	1	52	3.3461	0.0731
	score	772.5774	17574.8103	1	54	2.3738	0.1292
Backtracking vs optimized	time	2582.3915	39848.5955	1	52	3.3699	0.0721
	score	0	69.7113	1	54	0	1
Random vs optimized	time	0.0323	0.0515	1	52	32.5934	0
	score	772.5774	17574.8103	1	54	2.3738	0.1292

Table 4: Pairwise comparison of algorithm.

Each algorithms was applied to a sample of 30 pseudo-randomly generated maps where at least 1 algorithm was able to find a solution. Neither of maps belong to corner cases described in section 4.1 since they represent a small subset of all possible maps. Each map is 6-by-6, since gathering the data from backtracking on 10-by-10 maps will take ages (see *sparse_7* at table 3). There are no more than 10 orcs and human players in order not to make a map dense.
The data-set is present at table 5

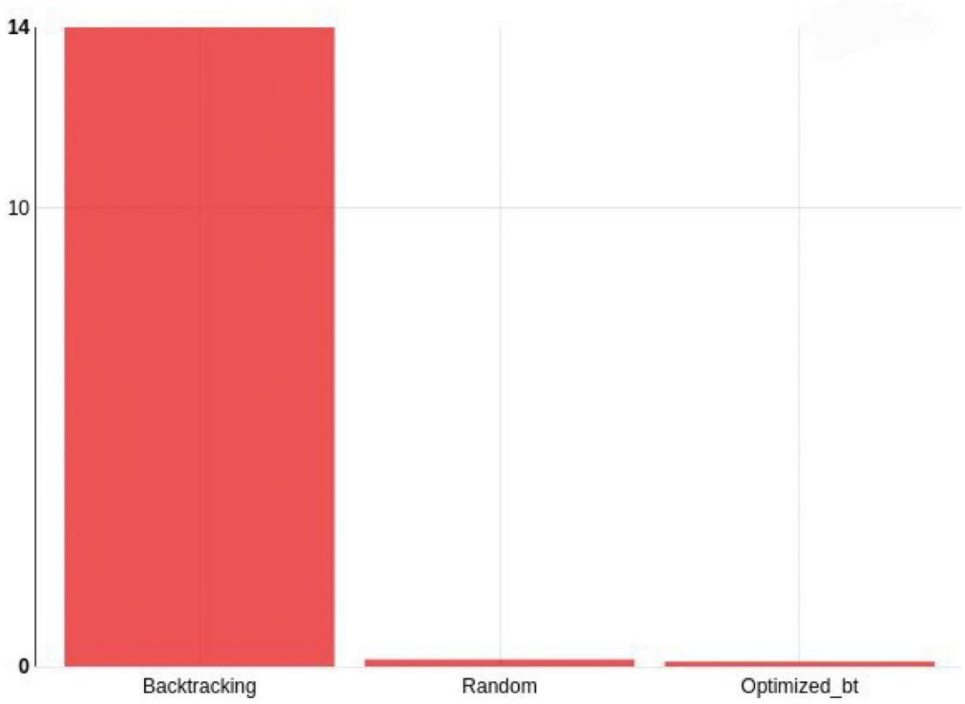

Figure 1: Mean time comparison.

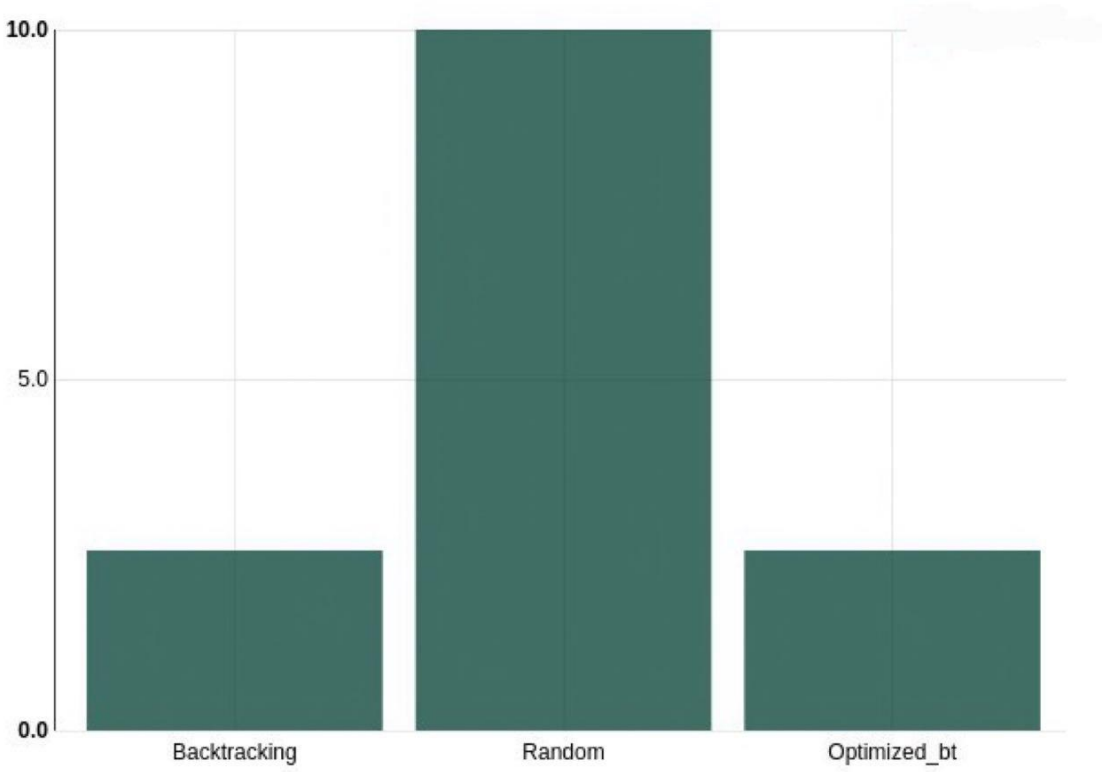

Figure 2: Mean score comparison.

By the results present in the table 4, we can definitely reject the null hypothesis for comparison between Random and optimized search in time for 6-by-6 maps. Rejecting for other comparison could be justified if I had better hardware or time for wait in order to calculate results for larger maps and have formal verification.

Informally we can estimate F-value for some comparison, making the following assumptions:

1. Random search with 100 tries and step limit of 100 terminates within 0.5 sec (based on run-time of data present in appendix B)

2. Both implementations of backtracking search eventually will find the shortest path - (no difference in final score)

3. Original backtracking executes either as fast as optimized or slower

The bigger map become, the more time is spent by both backtracking algorithm since computational complexity increases. Since the execution time is highly dependent on map design, there exists maps that are easy for backtracking implementations. The set of easy maps for optimized backtracking is bigger than set of easy maps for the simple one. Thus we can

roughly say that sum of squares between the groups will be much greater than within groups which will result in high F-value and rejecting null hypothesis for backtracking implementations' execution time.

Since random search's execution time is fixed by our assumptions, with growth of computational complexity, the total variance will also grow, resulting in high sum of squares within groups. Since random search's run-time is fixed, the standard deviation is close to zero, thus sum of squares within groups will be low. The same will hold for mean squares again resulting in high F-value.

As a result, we could possible reject all null hypotheses regarding execution time.

4.3 Random convergence

Table 5 stands for 30 6-by-6 maps. The table shows that random converges $\frac{27}{30} = 90\%$ of runs, which is very decent rate. After running a random search on a sample of 100 pseudo-random generated 20-by-20 maps with the same rules as in section 4.2, it turned out that convergence rate decreased to $\frac{52}{100} = 52\%$ (the data-set is present at appendix B).

As a result, random search is not a sound algorithm to apply in our problem unless information about map design is known (see section 4.1).

5 Improvement of vision

The main question is whether an ability to see 2 cell further rather than one improves the performance of algorithms. Yes, it will slightly improve. Consider situation when an agent is 2 cells away from the touchdown point. It can either directly move to the goal if there is no ork on the way or start exploring around the touchdown otherwise instead of possibly passing by the touchdown. Such behaviour will affect execution speed of all 3 algorithms.

Will such improvement help to solve previously unsolvable maps? For a backtracking algorithms a definite answer is no, since such algorithms guarantee to find the best solution if any exists due to traversing the whole search space. Speaking about the random search, if unsolvable maps where there is impossible to find the touchdown, then nothing changes. In other case, with extended vision the probability to make a correct decision on last 2 moves increases, so the probability to solve the map does, but very insignificantly.

Finally, the reverse question: will it make any previously solvable map unsolvable? No, since map's 'solvability' depends on it's design only. Having understood that we can proceed with the same reasoning as for the previous question.

A Data for statistical analysis

TestID	Backtracking		Random		Optimized backtracking	
	Score	Time (sec.)	Score	Time (sec.)	Score	Time (sec.)
0	1	0.03	1	0.16	1	0.13
1	1	0.02	1	0.14	1	0.05
2	3	0.06	7	0.15	3	0.15
3	3	0.06	4	0.15	3	0.13
4	4	0.57	7	0.15	4	0.09
5	2	0.03	2	0.16	2	0.05
6	2	195.49	2	0.17	2	0.07
7	1	0.11	1	0.15	1	0.08
8	4	7.19	No	0.15	4	0.11
9	1	0.04	1	0.17	1	0.04
10	2	0.04	2	0.17	2	0.17
11	3	32.03	4	0.15	3	0.15
12	1	0.03	1	0.15	1	0.09
13	4	21.70	7	0.16	4	0.17
14	4	1.49	4	0.15	4	0.10
15	2	0.04	2	0.15	2	0.14
16	4	0.03	4	0.09	4	0.14
17	4	66.84	4	0.16	4	0.12
18	3	0.49	5	0.17	3	0.07
19	No	0.02	No	0.12	No	0.18
20	2	0.09	2	0.16	2	0.08
21	1	0.28	1	0.16	1	0.11
22	4	0.05	No	0.16	4	0.06
23	3	1.33	3	0.16	3	0.08
24	2	22.33	2	0.15	2	0.08
25	3	0.10	3	0.16	3	0.09
26	2	0.03	2	0.16	2	0.09
27	3	15.91	5	0.16	3	0.22
28	2	0.05	2	0.14	2	0.04
29	4	10.2	4	0.14	4	0.15

Table 5: A comparison of score and elapsed time on 6-by-6 maps

B Random convergence

Table 6 show convergence of random on 30 10-by-10 maps.
The data below represent 100 20-by-20 maps.

```
No 2.txt| rand - 0.2409210205078125 sec
No 5.txt| rand - 0.24314403533935547 sec
No 6.txt| rand - 0.27263450622558594 sec
No 7.txt| rand - 0.213881254196167 sec
No 10.txt| rand - 0.2679882049560547 sec
No 11.txt| rand - 0.2332592010498047 sec
No 13.txt| rand - 0.24870681762695312 sec
No 14.txt| rand - 0.180342435836792 sec
No 16.txt| rand - 0.24698662757873535 sec
No 18.txt| rand - 0.17023992538452148 sec
No 19.txt| rand - 0.2351090908050537 sec
No 21.txt| rand - 0.23789596557617188 sec
No 22.txt| rand - 0.2111961841583252 sec
No 23.txt| rand - 0.22801423072814941 sec
No 27.txt| rand - 0.18100643157958984 sec
No 28.txt| rand - 0.16306662559509277 sec
No 31.txt| rand - 0.17395305633544922 sec
No 32.txt| rand - 0.20802783966064453 sec
No 36.txt| rand - 0.19962596893310547 sec
No 37.txt| rand - 0.23174214363098145 sec
No 43.txt| rand - 0.15204191207885742 sec
No 44.txt| rand - 0.24592185020446777 sec
No 45.txt| rand - 0.2526681423187256 sec
No 46.txt| rand - 0.25947141647338867 sec
No 47.txt| rand - 0.13971996307373047 sec
No 48.txt| rand - 0.23724865913391113 sec
No 50.txt| rand - 0.16917157173156738 sec
No 52.txt| rand - 0.22296524047851562 sec
No 53.txt| rand - 0.22559690475463867 sec
No 56.txt| rand - 0.23301362991333008 sec
No 59.txt| rand - 0.2338721752166748 sec
No 64.txt| rand - 0.18350768089294434 sec
No 65.txt| rand - 0.23682188987731934 sec
No 67.txt| rand - 0.2142772674560547 sec
No 68.txt| rand - 0.19294357299804688 sec
No 69.txt| rand - 0.22125005722045898 sec
No 70.txt| rand - 0.18365764617919922 sec
No 73.txt| rand - 0.22430109977722168 sec
```

```
No 75.txt| rand - 0.17740106582641602 sec
No 76.txt| rand - 0.28081488609313965 sec
No 77.txt| rand - 0.19503474235534668 sec
No 79.txt| rand - 0.2505202293395996 sec
No 83.txt| rand - 0.21472859382629395 sec
No 88.txt| rand - 0.19019842147827148 sec
No 89.txt| rand - 0.19520115852355957 sec
No 90.txt| rand - 0.21559810638427734 sec
No 92.txt| rand - 0.2631704807281494 sec
No 94.txt| rand - 0.1907057762145996 sec

Yes-10 34.txt| rand - 0.4282341003417969 sec
Yes-11 30.txt| rand - 0.2899479866027832 sec
Yes-12 4.txt| rand - 0.2645556926727295 sec
Yes-12 29.txt| rand - 0.24695539474487305 sec
Yes-12 54.txt| rand - 0.21273326873779297 sec
Yes-12 97.txt| rand - 0.24706649780273438 sec
Yes-14 66.txt| rand - 0.34903979301452637 sec
Yes-14 95.txt| rand - 0.25086402893066406 sec
Yes-15 3.txt| rand - 0.2602121829986572 sec
Yes-15 25.txt| rand - 0.3063502311706543 sec
Yes-15 51.txt| rand - 0.30806756019592285 sec
Yes-15 96.txt| rand - 0.2500147819519043 sec
Yes-16 78.txt| rand - 0.2322559356689453 sec
Yes-18 40.txt| rand - 0.2701857089996338 sec
Yes-19 35.txt| rand - 0.3061025142669678 sec
Yes-19 91.txt| rand - 0.271298885345459 sec
Yes-2 99.txt| rand - 0.2306373119354248 sec
Yes-22 0.txt| rand - 0.21579289436340332 sec
Yes-23 41.txt| rand - 0.2875680923461914 sec
Yes-26 1.txt| rand - 0.24782323837280273 sec
Yes-3 39.txt| rand - 0.24929165840148926 sec
Yes-3 72.txt| rand - 0.3705143928527832 sec
Yes-3 86.txt| rand - 0.34467363357543945 sec
Yes-36 61.txt| rand - 0.23972034454345703 sec
Yes-37 33.txt| rand - 0.3400270938873291 sec
Yes-4 8.txt| rand - 0.24848699569702148 sec
Yes-4 9.txt| rand - 0.1779329776763916 sec
Yes-4 49.txt| rand - 0.22096848487854004 sec
Yes-4 62.txt| rand - 0.23755812644958496 sec
Yes-4 71.txt| rand - 0.24361777305603027 sec
Yes-4 84.txt| rand - 0.2558414936065674 sec
```

```
Yes-4 85.txt| rand - 0.26599597930908203 sec
Yes-4 93.txt| rand - 0.2557523250579834 sec
Yes-42 20.txt| rand - 0.2380373477935791 sec
Yes-44 26.txt| rand - 0.3478822708129883 sec
Yes-44 74.txt| rand - 0.199171781539917 sec
Yes-47 63.txt| rand - 0.2545027732849121 sec
Yes-49 38.txt| rand - 0.2294003963470459 sec
Yes-5 12.txt| rand - 0.31253838539123535 sec
Yes-5 24.txt| rand - 0.26091456413269043 sec
Yes-5 42.txt| rand - 0.292147159576416 sec
Yes-5 87.txt| rand - 0.21729564666748047 sec
Yes-5 98.txt| rand - 0.2346515655517578 sec
Yes-55 82.txt| rand - 0.27967023849487305 sec
Yes-6 81.txt| rand - 0.2692081928253174 sec
Yes-68 17.txt| rand - 0.2393331527709961 sec
Yes-7 55.txt| rand - 0.2881937026977539 sec
Yes-7 57.txt| rand - 0.35713911056518555 sec
Yes-7 80.txt| rand - 0.22213172912597656 sec
Yes-77 58.txt| rand - 0.2521347999572754 sec
Yes-9 15.txt| rand - 0.2942485809326172 sec
Yes-9 60.txt| rand - 0.2339916229248047 sec
```

TestID	Random	
	Score	Time (sec.)
0	No	0.24
1	No	0.17
2	No	0.23
3	No	0.17
4	No	0.25
5	No	0.19
9	No	0.22
10	No	0.21
11	No	0.17
14	No	0.16
15	No	0.16
19	No	0.22
20	No	0.23
23	No	0.17
24	No	0.16
25	No	0.15
26	No	0.17
27	No	0.32
28	No	0.18
29	No	0.21
6	5	0.17
7	9	0.255
8	4	0.18
12	1	0.24
13	4	0.17
16	4	0.2
17	1	0.14
18	7	0.18
22	1	0.20
21	2	0.24

Table 6: A data for random convergence calculation on 10-by-10 maps