Design and implementation of a text-based game on a graph with reinforcement learning integration

Francesco Caporali, University of Turin

Contents

1 Introduction **1**

2 Design of the map **3**
2.1 Generation of the graph . . . 3
2.2 Adding the entities to the map . . . 8

3 Game parameters **11**
3.1 Compute possible pairs (stamina, life) . . . 11
3.2 Get game parameters . . . 14

4 The map class **16**

5 Gameplay **19**
5.1 Score . . . 22

6 Reinforcement learning player **25**
6.1 Monte Carlo reinforcement learning methods . . . 25
6.2 Rewards . . . 27
6.3 Implementation . . . 29
6.4 Considerations . . . 29

7 Computational costs **31**
7.1 Generation function . . . 31
7.2 Computation of stamina/life dictionary . . . 32

A Codes **34**

B Run the game **40**

References **41**

1

Introduction

The aim of this project is to introduce a new text-based game implemented upon a graph structure: `keep heading north`. Unlike the classical text-adventures we created a software which is built entirely on the game-map. We included the presence of hostile entities to fight or escape from (monsters), which represents one of the most popular features included in these kind of games.

The idea is to let the player move on a subset of a 2d grid normed with the infinite norm $(\mathbb{G}, \|\cdot\|_\infty)$, only choosing some selected points (vertices) and following predetermined paths (edges). We identify this structure as a directed graph and we add the weights naturally using the notion of norm inherited by the grid.
We report below an example of graph generated following this prototypical idea.

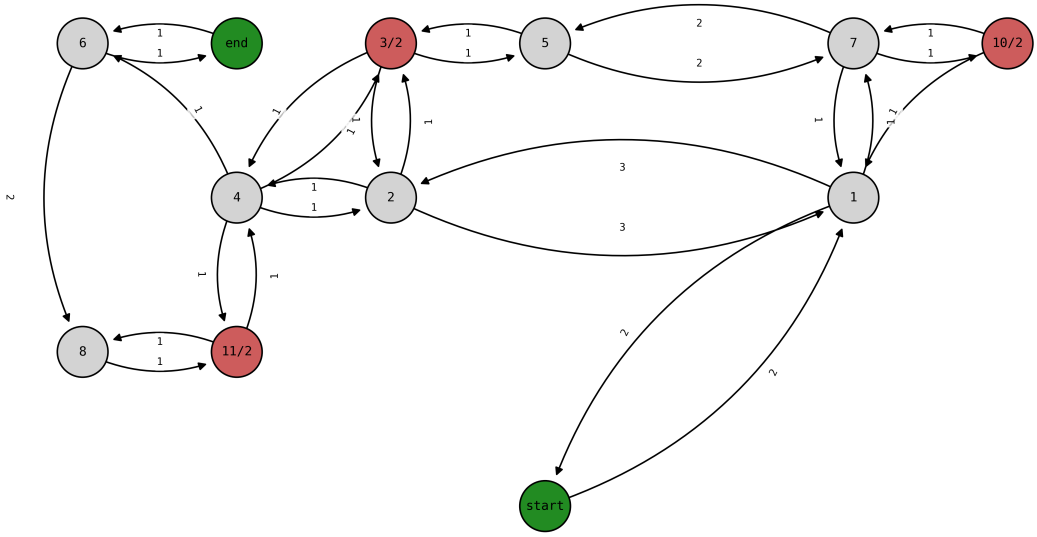

Figure 1.1: Example of map generated algorithmically where the player will move. The nodes in green correspond to the start and the end of the map whereas the red nodes are related to the presence of hostile entities.

The implementation has been realized through a customized version of a graph class with several additional features which make the game more ralistic and entertaining. The construction is fully randomic hence every session will be different from the previous one.
During each phase of the gameplay the player makes choices by looking at the portion of

the map which has been discovered until that moment. Each action is recorded and drawn allowing them to take decisions consciously.
During their travel they will encounter monsters and dead ends. Facing all these difficulties and exploring the cave they will have to try not to lose all their energies and life points until they will finally reach the end.

After the actual game experience a total score is visualized allowing the player to assess the quality of their session. At the very end there is the possibility to visualize a perfect match computed through an exact algorithm and then an entire game session played by a reinforcement learning agent trained by playing on the same game map.
Several graphical details have been implemented but without paying too much attention to the portability of the game (see Appendix B).

2

Design of the map

We decided to implement the game using a directed weighted graph because these objects are one of the most intuitive way to schematize a map. Indeed the nodes (or vertices) can be used to represent locations and the weighted edges allow us to describe the possible ways to move between them, also taking note of their relative distances. Hence we exploited this basic data structure as main tool in analogy with the primitive framework in which the player lives. The protagonist wakes up in a unfamiliar place and needs to find a way to keep track of where they go by just using pen and paper.

2.1 Generation of the graph

One of the first problem we had to face was the creation of the graph. We wanted the game to be replayable and also every session to be completely new. So the goal was to design an algorithm to generate randomly some directed graphs keeping in mind several constraints:

1. the algorithm has to take a parameter (`bound`) as input and to construct a graph whose size (number of nodes) depends on that parameter;
2. the edges of the graph must not overlap each other except at the coordinates where they start and end;
3. a `start` and an `end` point have to be set, preferably as distant as possible;
4. there must always be a way to move from a fixed vertex `v` to the `end`;
5. the graph should not have too many vertices;
6. the graph has to be fully connected.

The proposed solution is reported in Pseudocode 1 in an easily readable syntax.
The function `generate` builds a graph `g` defining the queue `visited` where the nodes, which will be starting points of new edges, are stored. During each iteration of the main loop a vertex `u` is popped by the queue and a set of `n_neighours` edges starting from `u` is generated by the second internal loop.
The possible edges are created choosing randomly one of the possible 8 cardinal or intercardinal directions and an integer number between 1 and 3 representing their lenght. Hence there are a total of 24 possible choices.
Here we also check some correctness conditions (lines $13 - 16$) which will be detailed below. After having generated a correct edge we insert it in the graph and in case we encountered a new vertex we also insert it in `g` and `visited`.

Remark 2.1. The most internal loop has an high probability to terminate very fastly. Indeed in the worst case there is only one possible correct direction to choose among the 24 available from `u`. This could happen if we reached the maximum number of vertices (`max_vertex`) and we are only allowed to travel backwards through the edge which originally created `u`.
In this case the probability of doing the wrong choice repeatedly becomes very low. Let $p = \mathbb{P}(\text{choose an incorrect vertex}) = \frac{23}{24}$, then by selecting 50 times a possible direction we have that the probability of being still stuck in the loop is

$$\mathbb{P}(\text{stuck in loop}) = \mathbb{P}(\text{choose an incorrect vertex})^{50} = p^{50} \approx 0.96^{50} \approx 0.12.$$

Pseudocode 1 Algorithm to generate the underlying graph of the game map.

```
1:  function generate(bound)
2:      g = empty directed weighted graph
3:      max_vertex = (2 · bound + 1)^2 // 4
4:      insert start in g
5:      visited = [start]
6:      while visited is non-empty do
7:          u = pop(visited)
8:          n_neighbours = random in [1, 2, 3]
9:          for _ in 1, ..., n_neighbours do
10:             while true do
11:                 edge = choose a random edge
12:                 v = arrival vertex of edge
13:                 if v is outside the boundaries then continue
14:                 if opposite of edge is already in g then break
15:                 if v is new and we already have max_vertex vertices in g then continue
16:                 if edge does not cross any already taken node in g then break
17:             if v is new then
18:                 insert v in g
19:                 append v to visited
20:             insert edge in g
21:             insert opposite of edge in g with probability 0.5
22:     end = last node found in bfs(g, start)
23:     for u in vertices(g) do
24:         if end is not found during dfs(g, u) then
25:             restart generate(bound)
26:     return g
```

In order to handle the parameter `bound` (condition 1) the algorithm sets an initial maximum size of the map, which is $2(2 \cdot \texttt{bound}) + 1$. The idea is to create a squared grid of this size, keeping in mind that the vertices of our graph will only lie in points with even coordinates (i.e. (a, b) s.t. $a, b \in 2\mathbb{Z}$). With line 13 of Pseudocode 1 we check that the vertices are placed inside this grid.
To clarify what we just described we report a picture of the grid that would be considered with $\texttt{bound} = 2$ (see Figure 2.1 left side).

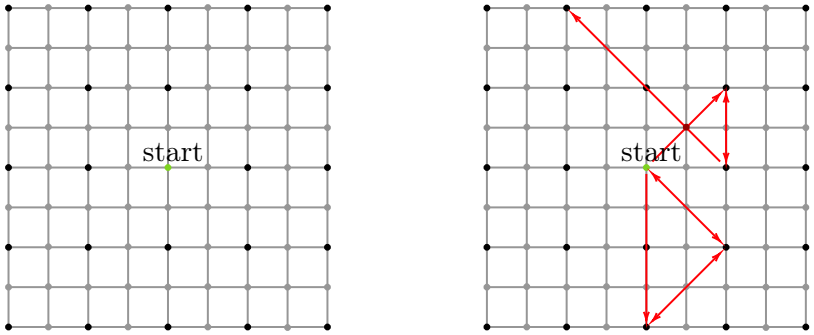

Figure 2.1: Grid considered by the generation algorithm with `bound` $= 2$ on the left. Possible error if the gray bullets are not taken in account on the right.

The coordinates where the vertices could be positioned are marked in black (denoted as `integer nodes` inside the source code).
The other bullets, marked in gray (denoted as `half nodes`), are necessary for our implementation because allow us to avoid distinct edges to overlap each other in points which are not vertices of the graph (part of condition 2). An example of possible mistake that could occur if those half nodes are not considered is reported in Figure 2.1 right side.

Remark 2.2. Note that the `half nodes` (a, b) s.t. $a \in 2\mathbb{Z}$ or $b \in 2\mathbb{Z}$ cannot be points where two edges generated choosing cardinal or intercardinal directions can overlap each other.

This condition is controlled through a sparse matrix (`location` in the source code) in which we mark the crossed coordinates and the visited ones as follows:

- 0, not visited nodes;
- 1, occupied nodes (vertices);
- 2, visited nodes (edges).

The chunk of code where we update the `location` matrix is reported below. It deals with the computation of the entries of the matrix to change, which are retrieved from the directions and the coordinates of the vertices.

```
# update of the location matrix
for i in range(1, step):
    # half nodes crossed by an edge
    location[(prev_x + direction[0] * i + bound) * 2 - direction[0], (prev_y +
 ↪ direction[1] * i + bound) * 2 - direction[1]] = 2
    # nodes crossed by an edge
    location[(prev_x + direction[0] * i + bound) * 2, (prev_y + direction[1] * i +
 ↪ bound) * 2] = 2
# last half node crossed by an edge
location[(prev_x + direction[0] * step + bound) * 2 - direction[0], (prev_y +
 ↪ direction[1] * step + bound) * 2 - direction[1]] = 2
# last node occupied by a vertex
location[(prev_x + direction[0] * step + bound) * 2, (prev_y + direction[1] * step +
 ↪ bound) * 2] = 1
```

Listing 2.1: Portion of the function `generate`: update of the `location` matrix.

Note that with the 1's we also prevent the edges to overlap on other vertices, hence we fulfilled the condition 2.
The sparsity of the matrix is introduced because in this way only the crossed or visited nodes occupy space in memory.
The key lines of the function where we check these conditions are reported below in Listing 2.2. They correspond to the implementation of the line 16 of the Pseudocode 1.

```
# check correctness edge:
flag = True
# the flag is true if:
#   the edge never passes on already taken nodes in the map (1 or 2 in location),
#   the last half node is analyzed separately
#   (check both integer and half nodes)
for i in range(1, step):
    # half nodes
    flag = flag and (location[(prev_x + direction[0] * i + bound) * 2 - direction[0],
 ↪ (prev_y + direction[1] * i + bound) * 2 - direction[1]] == 0)
    # integer nodes
    flag = flag and (location[(prev_x + direction[0] * i + bound) * 2, (prev_y +
 ↪ direction[1] * i + bound) * 2] == 0)
# last half node
flag = flag and (location[(prev_x + direction[0] * step + bound) * 2 - direction[0],
 ↪ (prev_y + direction[1] * step + bound) * 2 - direction[1]] == 0)
# same check for the last node taking in account that it is allowed to end in an
 ↪ already taken node (if it is a vertex), but not on an edge
flag = flag and (location[(prev_x + direction[0] * step + bound) * 2, (prev_y +
 ↪ direction[1] * step + bound) * 2] in [0, 1])
```

Listing 2.2: Portion of the function `generate`: check existence of overlaps between edges.

The line 4 of the Pseudocode 1 sets up the starting vertex which is always positioned at coordinates $(0, 0)$ (entry $(2 \cdot \texttt{bound}, 2 \cdot \texttt{bound})$ of the matrix `location`). Similiarly the line 22 is related to the ending vertex, which is computed as the last vertex found by a `bfs` visit of the graph starting from `start`[1]. This approach is motivated by the fact that the `bfs` explores the graph classifying each node according to its distance (minimum number of edges that separate them) from the source, hence the farthest vertices are all in the last visited layer. What we just discussed makes the algorithm satisfy the condition 3.

From line 23 to 25 we deal with condition 4, in particular through a sequence of `dfs` visits we ensure that there is a way to reach the `end` starting from any inner vertex.
Actually if this condition is not satisfied we restart the generation of the map. Anyway the termination of our procedure is probabilistically guaranteed. Indeed looking at 1000 randomic generations of graphs through our function, using different values of the parameter `bound`, we can observe that the number of attempts required to obtain an outcome which satisfies our condition is almost always very low.
We reported the results in the following Figure 2.2.

[1] In practice this computation is implemented in the main class of the game as an independent method named `compute_end`.

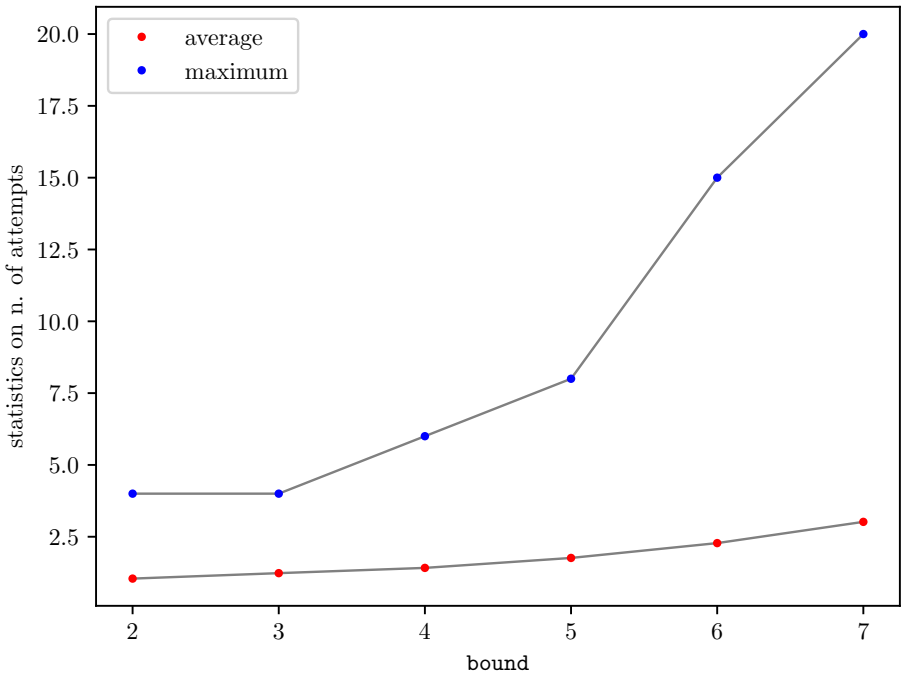

Figure 2.2: Maximum and average number of attempts required to obtain an outcome which satisfies the condition 4, on a total of 1000 generations ($\texttt{bound} = 2, \ldots, 7$).

Actually we expected the probability to generate a faulty graph to be very low indeed we incresed the number of connections by adding the possibility of generating the backward edge every time we add a new edge (see line 21 of the Pseudocode 1).

The condition 5 follows immediately by the introduction of the parameter `max_vertex` and by its use at line 15.

Remark 2.3. Actually we also impose a lower bound on the number of entities to be inserted in order to obtain non-trivial graphs[2]. We set this lower bound to $(2 \cdot \texttt{bound} + 1)^2 \,//\, 5$.

Finally the last condition (6) is a consequence of the method used to choose the new vertices as arrival points of the newly introduced edges.

Remark 2.4. The function generate must take as input a value of $\texttt{bound} \geq 2$. Indeed with $\texttt{bound} = 1$ it is very likely (probability 2/3) to enter an infinite loop trying to add more than one neighbour to the starting vertex. This happens because the value `max_vertex` is equal to 2.

To allow a deeper comprehension of the different phases of construction of the game we introduce a driving example. For this purpose we set a seed for the randomicity which will be mantained during the following chapters.
First we show a representation of the output graph `g` returned by the function `generate` (Figure 2.3). It is realized using a custom function `draw` defined using various macros from the package `networkx`.

[2]This possibility occurs very rarely hence we decided not to analyse it deeply.

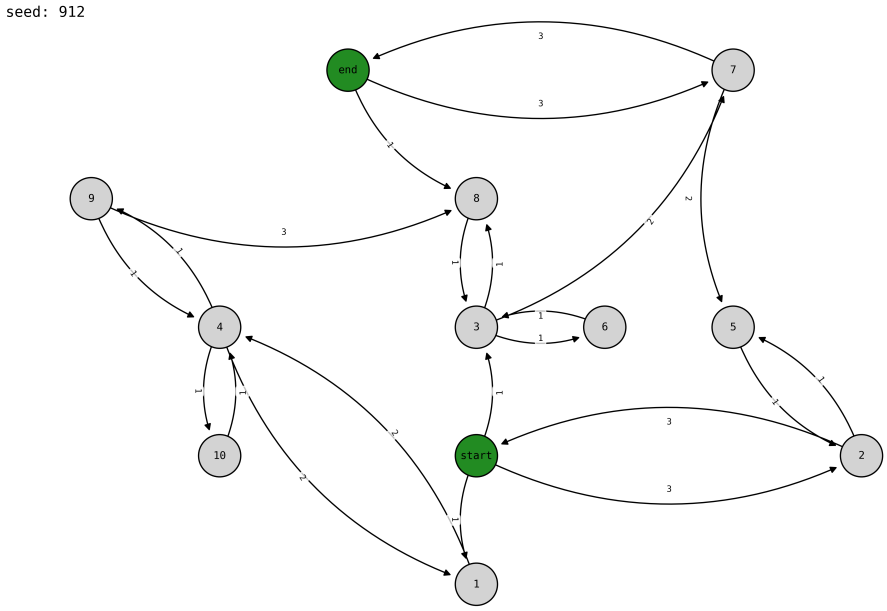

Figure 2.3: Main example: representation of the output returned by the function `generate`.

2.2 Adding the entities to the map

After having generated the underlying graph structure of the map we had to place some hostile entities on some of the nodes. In this case the problem is easier. First we had to decide how many entities would have been present on the map, then we had to insert them trying to respect the two constraints which appear necessary:

1. `start` and `end` should be entity-free;
2. two adjacent entities should be avoided if possible.

The implemented solution follows the Pseudocode 2.

Pseudocode 2 Algorithm to insert the entities on the game map `m`.

```
1:  function add_entities(m)
2:      num_entities = num_vertices(m) // 5 + 1
3:      white_list = full_list = vertices(m) \ [start, end]
4:      for _ in 1, ..., num_entities do
5:          if white_list is non-empty then
6:              entity = random in white_list
7:              remove entity from white_list
8:              for neighbour in neighbours(entity) do
9:                  remove neighbour from white_list
10:         else
11:             entity = random in full_list
12:         remove entity from full_list
13:         add_entity entity to m with random power in [1, 2, 3]
```

We decided to introduce approximatively one entity every 5 nodes. This number is only motivated by the aesthetic result given by the code. We report the average number of entities added to the maps for different values of the parameter `bound`.

`bound`	2	3	4	5	6	7
min	2	2	4	5	7	10
max	2	3	5	7	9	12
average	2	2.99	4.98	6.98	8.98	11.98

Table 2.1: Minimum, maximum and average number of entities generated. The statistic is computed on a total of 1000 generations (`bound` $= 2, \dots, 7$).

We introduced a `white_list` of nodes where it should be possible to insert entities, and a `full_list` containing the nodes which are entity-free until then. Both the lists are initialized with all the vertices except for the `start` and the `end` node, according to the first condition introduced above.
During each iteration of the main loop we try to insert an entity in a randomic vertex extracted from the `white_list` until it is possible. After the extraction we remove the entity node from both the `white_list` and the `full_list`, moreover we delete all its neighbours from the `white_list`. This allows us to guarantee the second condition we asked for as long as we can. When the `white_list` gets empty it is no more possible to spread out the entities but thanks to the `full_list` we keep placing them randomly on the map.
In the last line we assign a `power` parameter to each entity. This will be important in the subsequent functions in order to create a more entertaining gameplay.

Remark 2.5. It is worth noting that, defining $n =$ **num_vertices**(`m`), it is always possible to insert $n \,//\, 5 + 1$ entities if the underlying graph of `m` has at least 3 nodes. Indeed it holds

$$n - 2 \geq n \,//\, 5 + 1 \iff n - \lfloor n/5 \rfloor \geq 3 \implies n \geq 3,$$

but also, since $n - \lfloor n/5 \rfloor \geq n - n/5$, we have

$$n \geq 4 \implies n \geq 15/4 \iff \frac{4}{5} n \geq 3 \iff n - n/5 \geq 3 \implies n - \lfloor n/5 \rfloor \geq 3.$$

Finally for $n = 3$, $n - 2 = 1 \geq 1 = n \,//\, 5 + 1$.
By recalling that `m` has at least $(2 \cdot \texttt{bound} + 1)^2 \,//\, 5$ vertices with `bound` ≥ 2 we have that **num_vertices**(`m`) ≥ 5.

The function `add_entities` adds new information to our map `m`. This new layer can be observed on the main example previously introduced. See Figure 2.4.

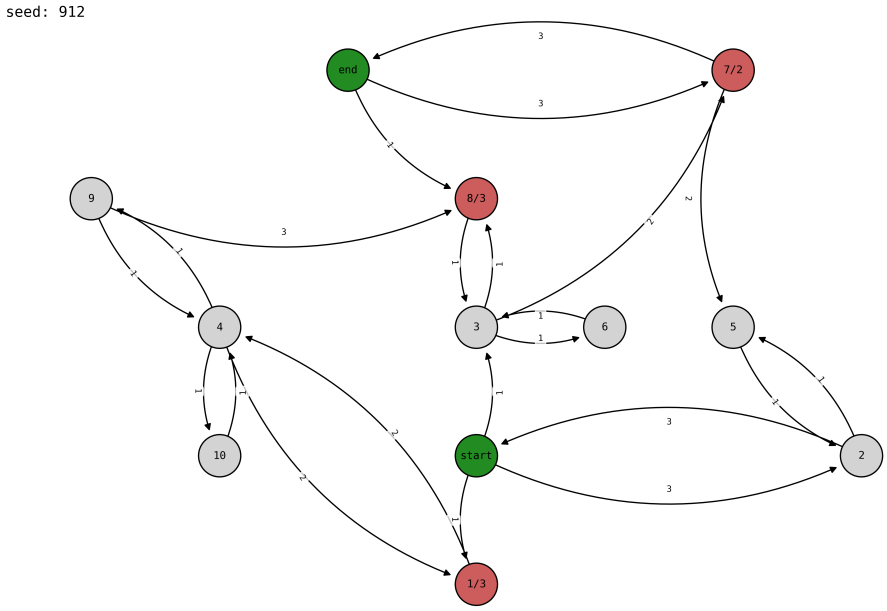

Figure 2.4: Main example: integration of the entities to the map `m` through the function `add_entities`.

3

Game parameters

After having generated a map as described in the Chapter 2, the following natural step consists in introducing the main game parameters.
As almost every role playing game we introduced two basic parameters: stamina and life. The idea was to create an adventure which puts the player in the starting point of our map and let them explore.
During this phase the energy of the protagonist keeps decreasing due to the fatigue, moreover if they encounter some hostile entities they also will lose some of their life points while defeating it, proportionally to the power of the monster (one `life` point for each `power` point). These two parameters will determine the survival or death of the player.

3.1 Compute possible pairs (stamina, life)

In order to set up the initial values of stamina and life we decided to explore all the *optimal* routes from the `start` to the `end`.
First we needed to introduce a definition of *optimality*. In a quite natural way we decided to divide all the possible winning paths with respect to the amount of life points lost during the traversal.
With this classification in mind we realized an algorithm (see Pseudocode 3) to find an optimal path, in terms of stamina cost, for each possible value of life consumption which can be obtained.

To partition the set of possible routes we tried to simplify the problem. Instead of looking for the paths from `start` to `end`, the idea was to search for all the optimal ways to travel from each entity to any other, possibly without encountering some specified entities. These sets of avoided entities are referred to as blacklists.
To do so, the algorithm loops among the possible partial permutations `p` of the set of entities. For each `p`, it tries to find the optimal way to cross all and only the entities in `p` with order. Hence, for every considered instance, it has to perform a sequence of shortest path computations using a `dijkstra`[1] algorithm modified to avoid a set of blacklisted nodes. Initially it is computed the shortest path between `start` and $\texttt{e}_1$ (the first entity) with a `blacklist` composed by all the remaining entities. Then the procedure does the same from $\texttt{e}_1$ to the second entity $\texttt{e}_2$ with `blacklist` = **entities**(`m`) $\setminus$ [$\texttt{e}_1$, $\texttt{e}_2$]. It goes on analogously until the `end`.

[1] A version of the Dijkstra algorithm (`_dijkstra`) is included as a protected method in the main class of the game.

Pseudocode 3 Computation of optimal values for stamina and life.

```
1: function compute_stamina_life(m):
2:     optimal_path = {}
3:     stamina_life = {}
4:     for p partial permutation of entities(m) do
5:         life, stamina, path = 0, 0, [ ]
6:         p = start + p + end
7:         blacklist = entities(m) + end
8:         for u in p do
9:             v = successive of u in p
10:            blacklist = blacklist \ v
11:            if (u, v, blacklist) not in optimal_path then
12:                current_cost, current_path = dijkstra(u, v, blacklist)
13:                for s subset of entities(m) \ current_path do
14:                    optimal_path[(u, v, s + blacklist)] = current_cost, current_path
15:            else
16:                current_cost, current_path = optimal_path[(u, v, blacklist)]
17:            stamina += current_cost
18:            life += power(u)
19:            path += current_path
20:        if stamina < ∞ then
21:            if life not in stamina_life or
                   (life in stamina_life and stamina < stamina_life[life][0]) then
22:                stamina_life[life] = (stamina, path)
```

Clearly this problem is extremely expensive indeed if $|\textbf{entities}(\texttt{m})| = n$ then the number of partial permutations to consider is $\sum_{k=0}^{n} \frac{n!}{(n-k)!}$. We report in the following table the number of partial permutations for the main values of interest (reported in Table 2.1):

entities	2	3	5	7	9	12
partial permutations	5	16	326	13700	986410	1302061345

Table 3.1: Number of partial permutations of different values: we used the rounded values of the average number of entities for `bound` $\in [2, \ldots, 7]$.

In order to reduce the complexity of the problem, while analyzing all the possible partial permutations, we construct a dictionary where we store all the results of the `dijkstra` algorithm. This dictionary can be used to skip a large part of all the shortest path computations by noting that if we apply `dijkstra` to the triple (`u`, `v`, `blacklist`) we also know that the resulting `path` is optimal for every triple (`u`, `v`, `s` + `blacklist`) with `s` subset of **entities**(`m`) \ `path`. This is because if we blacklist other entities which were already avoided by the best path found from `u` to `v`, the shortest path will necessary remain unchanged.
The last lines of the inner loop are related to the update of the three main variables of the algorithm.

Finally, in the chunk $20 - 22$, we store the couple (`stamina`, `path`) inside the dictionary `stamina_life` in correspondance with the entry `life`, but only if `stamina` represents a new optimal value for its life category. In particular with the line 20 we ensure that there is actually a way to cross the map following the order given by the permutation `p`. Indeed it is remarkable that in several cases the topology of the graph and the arrangement of the entities lead to unreachable destinations.

Remark 3.1. The optimal paths between the pairs `u`, `v` $\in$ `full_list` such that `u` $\neq$ `end`, `v` $\neq$ `start` and `u` $\neq$ `v` can be precomputed in order to be reused in the loop. This is because they create many different triples in the dictionary `optimal_path`, but most importantly more entries than the ones which would be created in the first iterations of the loop.
Indeed, if we compute the shortest path between `u` and `v` with a `blacklist`, there could happen two distinct things:

- the set of avoided entities (`entities` $\setminus$ `current_path`) at line 13 coincides with the one created by `dijkstra` between the same nodes without a `blacklist`. In this case the number of entries added to the dictionary `optimal_path` are less than the ones which could have been precomputed;
- the set of avoided entities is bigger than the one created by `dijkstra`. In this case the set of added entries is disjoint from the set added by the precomputation.

Note that the full permutations of the entity list always require the triples defined in the precomputation in order to skip the lines $12 - 14$, hence it is always convenient to perform the precomputation.

Actually the lines $13 - 14$ are optimized through a second dictionary used to map all the triples $(\texttt{u}, \texttt{v}, \texttt{s} + \texttt{blacklist})$ to $(\texttt{u}, \texttt{v}, \texttt{blacklist})$ so that the potentially heavy entry `path` is not stored more than one time.
We report in the Listing 3.1 the translation in `Python` of the lines $11 - 14$.
If one looks carefully at the implementation it can be observed that `s` is any possible subset of **entities**(`m`) $\setminus$ `current_path` $\setminus$ `blacklist` and not of **entities**(`m`) $\setminus$ `current_path`. The reason is that in this way we perform less iterations since the `blacklist` will eventually be readded to `s`.

```
1  # if we haven't got the shortest path for the triple (node_1, node_2, blacklist) we
    ↪ compute it and store it (we also update pairs_alias)
2  if (node_1, node_2, tuple(blacklist)) not in pairs_alias:
3      pairs[(node_1, node_2, tuple(blacklist))] = self._dijkstra(node_1, node_2,
    ↪ blacklist)
4      current_cost, current_path = pairs[(node_1, node_2, tuple(blacklist))]
5      for alias in powerset(set(entities) - set(pairs[(node_1, node_2,
    ↪ tuple(blacklist))][1]) - set(blacklist)):
6          pairs_alias[(node_1, node_2, tuple(list(alias) + blacklist))] = (node_1,
    ↪ node_2, tuple(blacklist))
```

Listing 3.1: Portion of the function `compute_stamina_life`: compute the shortest path between `u` and `v` with a `blacklist` if needed and add it to `optimal_path`.

The output of the function `compute_stamina_life` on the running example is the dictionary schematically reported in Table 3.2.

key: life	value[1]: stamina	value[2]: path
2	6	$[0, 3, 7, 11]$
5	8	$[0, 3, 8, 3, 7, 11]$
8	13	$[0, 1, 4, 9, 8, 3, 7, 11]$

Table 3.2: Main example: dictionary containing the optimal values for the stamina found for each possible life consumption, the optimal path is reported too (output of `compute_stamina_life`).

3.2 Get game parameters

At this point of the implementation we have a raw version of all the game parameters. The subsequent step was to refine the values in the `stamina_life` dictionary according to the game settings.
For this purpose we introduced a `gamemode` parameter which allows the player to choose one of three different possibilities: `survivor`, `explorer` and `balanced`. Then, according to the choice we set up the initial values of `stamina` and `life`.
Each gamemode corresponds to a different strategy to face the text-adventure:

- in `survivor` mode the objective is to find the `end` trying to avoid the hostile entities;
- in `explorer` mode the player has to explore the cave searching for as monsters as possible and battling them;
- in `balanced` mode the strategy is less extreme, the idea is to find a good compromise between exploring and preserving the `stamina`.

Obviously in every mode it is given a different priority to the consumption of `stamina` and `life`. In particular for `explorer` and `balanced` modes the idea is that every monster defeated gives an `experience` reward equal to the `life` lost. So the more `experience` is acquired the best the session will be valued, always taking in account that in `balanced` mode the `life` consumption must be managed carefully. Following these guidelines we implemented the algorithm `get_parameters` reported in Listing 3.2.

```
def get_parameters(self, gamemode):
    """
    description:
        Return a list with life, stamina and their corresponding path associated to
 ↪ the gamemode.
    syntax:
        parameters = m.get_parameters(gamemode)
    """
    lives = list(self._stamina_life.keys())
    lives.sort()
    if gamemode  == "balanced":
        life = lives[len(lives) // 2]
    elif gamemode == "survivor":
        life = lives[0]
    elif gamemode == "explorer":
        life = lives[-1]
    stamina = self._stamina_life[life][0]
    path = self._stamina_life[life][1]
    return [life, stamina, path]
```

Listing 3.2: Function `get_parameters`.

Note that the output of the code is just one of the optimal triples stored in the dictionary. This does not mean that we expect the player to win the game only if he plays perfectly. Indeed after the computation of the couple (`best_stamina`, `best_life`) for the selected `gamemode` we multiply it by a modifier parameter `difficulty`. We have foreseen two distinct `difficulty` choices: `easy` and `hard`. In both cases the set up of the couple corresponding to the initial game parameters (`stamina`, `life`) follows this easy rule:

$$(\texttt{stamina}, \texttt{life}) = (\texttt{round}(m \cdot \texttt{best\_stamina}), \texttt{round}(m \cdot \texttt{best\_life})).$$

The value of m is set to 3 for the `easy` mode and to 1.5 for the `hard` mode.

For completeness we report in Table 3.3 the outputs of the Listing 3.2 associated to the various gamemodes and difficulties for our main example.

gamemode	survival easy	survival hard	balanced easy	balanced hard	exlporer easy	explorer hard
life	6	3	15	8	24	12
stamina	18	9	24	12	39	20

Table 3.3: Main example: output of the function `get_parameters` for the various gamemodes and difficulties selectable.

4

The map class

The source code is entirely based on a core class which allows us to manage the design and construction of the map, the computation of game parameters and the on-screen printing of the output (map and additional parameters).
We report the structure of the class in a schematic way in Figure 4.1.

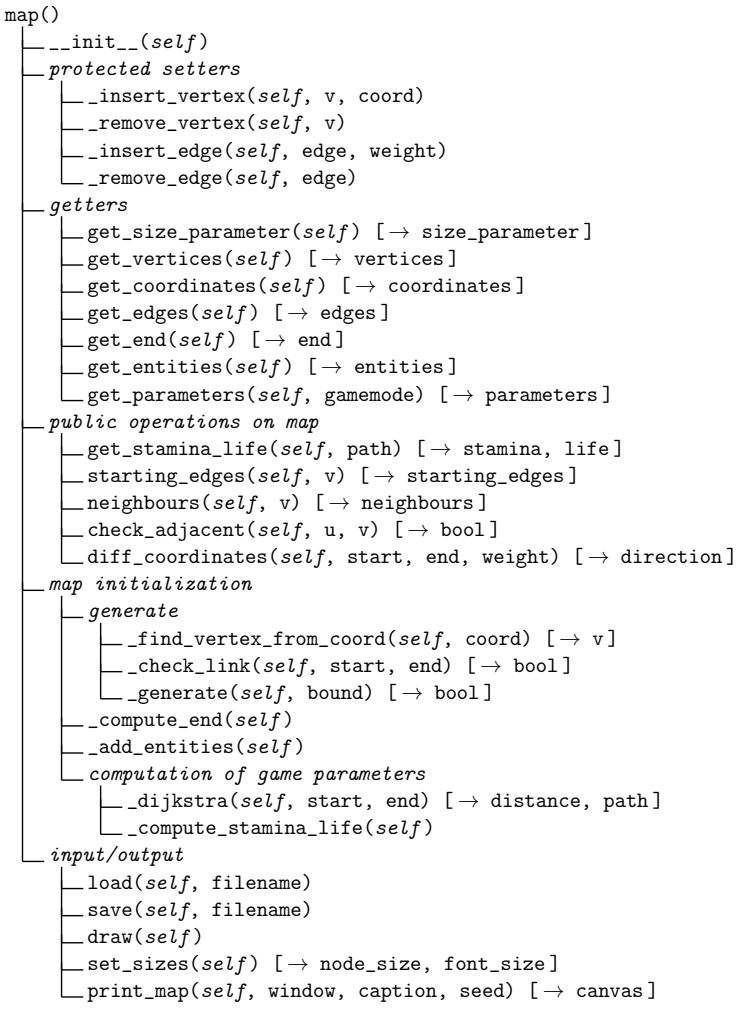

Figure 4.1: Structure of the class `map`.

The methods of the class are divided in 5 subgroups by their type and scope.

1. The *protected setters* are 4 basic procedures that work on the underlying graph structure to add/remove edges and vertices. They are called by the construction methods, i.e. `_generate` and `load`.
2. Several *getters* have been implemented in order to access the 6 protected instance attributes defined inside the `__init__` procedure. The first 6 getters deal with all the main dictionaries, except for `_stamina_life` which is handled differently. The last one is the only non-trivial getter, indeed, as previously explained it takes a `gamemode` as input and extracts an entry of `_stamina_life` according to it.
3. Some additional *public operations on the map* have been added in order to avoid any necessary external manipulation of the protected areas of the class. The only two functions which deserve a brief explanation are `get_stamina_life`, which retrieves the stamina and life consumption obtained traversing a path, and `diff_coordinates`, which computes the *direction* of an edge (as unitary vector in uniform norm).
4. The *map initialization* functions have been widely discussed in Chapters 2 and 3. Except for `_find_vertex_from_coord` and `_check_link` which are auxiliary functions, all the other methods of this group correspond to their previously introduced homonymous.
5. The last group, *input/output*, is a collection of load, save and representation methods. The most important procedure in this group is `draw`: it is a complex function that allows 13 optional parameters through which it can realize highly customizable representations of the generated maps.

The main instance attributes of the class are initialized in the lines $8-13$ of the constructor `__init__` reported in Listing 4.1. Note that the parameter `_size_parameter` corresponds to the variable `bound` used by the `generate` algorithm introduced in Chapter 2.
Almost all these attributes are implemented as dictionaries because of the time-complexity advantages given by this data structure. In particular we will use search, removal and insertion which will all cost $\mathcal{O}(1)$.

```
1  def __init__(self, generate = 0):
2      """
3      description:
4          Inizialization of an empty map (the map() class implements a graph structure
   ↪ with dictionaries with several additional specifications).
5      syntax:
6          m = map()
7      """
8      self._size_parameter = generate
9      self._coordinates = {}
10     self._vertices = {}
11     self._end = None
12     self._entities = {}
13     self._stamina_life = {}
14     if generate != 0:
15         flag = False
16         while not(flag):
17             self._coordinates = {}
18             self._vertices = {}
19             self._entities = {}
20             flag = self._generate(generate)
21         self._add_entities()
22         self._compute_stamina_life()
```

Listing 4.1: `__init__` function of the class `map`.

The initialization constructor allows both the automatic generation of a map made through the functions `_generate`, `_add_entities` and `_compute_stamina_life` and the loading of a pre-saved map which can be accomplished through the command `load`.

5

Gameplay

Until now we described how we built the environment where the game has been developed. The following step was the actual build-up of the game session. The key steps of a session are all called with order inside the loop of the `main` function (reported in Listing 5.1).

```
def main():
    save = []
    print_title()
    while True:
        # generate map
        m, seed = generate_map()
        # generate parameters
        gamemode, difficulty, best_life, best_stamina, best_path, life, stamina =
 ↪ generate_parameters(m)
        # start the game
        window, canvas, current_life, current_stamina, counter_games = start_game(m,
 ↪ life, stamina, gamemode, seed, save)
        # end of the game
        show_results(life, current_life, best_life, stamina, current_stamina,
 ↪ best_stamina, counter_games, gamemode, difficulty)
        # show optimal solution
        canvas = show_optimal_solution(m, window, canvas, seed, best_life,
 ↪ best_stamina, best_path, save)
        # let an rl algorithm play the same game
        rl_solution(m, window, canvas, seed, stamina, life, best_stamina, best_life,
 ↪ gamemode, difficulty, save)
        # play again
        restart = play_again()
        if restart:
            if not(notebook):
                window.destroy()
        else:
            break

notebook = False
if __name__ == "__main__":
    main()
```

Listing 5.1: `main` function of the game.

The first function is related to the construction of a map `m`, it basically calls the `__init__` of the class `map` after having requested the `bound` value to use as input. Similarly, the subsequent function `generate_parameters` asks for the `gamemode` and the `difficulty` then returns the modified output of the method `get_parameters`.
What follows is the function `start_game` which is schematized in the Pseudocode 4 below and therefore does not need any further explanation:

Pseudocode 4 Scheme of the function `start_game`.

```
1: function start_game(m, life, stamina, gamemode)
2:     intro_game(gamemode)
3:     counter_games, visited_nodes, visited_edges = 0, [0], []
4:     while True do
5:         counter_games += 1,
6:         current_position, current_life, current_stamina = 0, life, stamina,
7:         active_entities = entities(m)
8:         print the current state using m.print_map(_)[1]
9:         while True do
10:            current_position = choose_direction(m, current_position)
11:            update of current_stamina
12:            if current_stamina < 0 then
13:                if the player chooses to restart then break
14:                else goto line 34
15:            update of visited_nodes and visited_edges
16:            print the current state using m.print_map(_)
17:            if current_position in active_entities then:
18:                if the player can go back and chooses to escape then
19:                    current_position = past_position
20:                    update of current_stamina
21:                    if current_stamina < 0 then
22:                        if the player chooses to restart then break
23:                        else goto line 34
24:                    update of visited_edges
25:                else
26:                    update of current_life
27:                    if current_life < 0 then
28:                        if the player chooses to restart then break
29:                        else goto line 34
30:                    remove current_position from active_entities
31:                print the current state using m.print_map(_)
32:            if current_position == end then
33:                goto line 34
34:    return current_life, current_stamina, counter_games
```

Remark 5.1. The function `print_map` uses the standard `Python` interface through the package `tkinter`. At the very beginning of the game session a `window` is created and at every call of `print_map` a `canvas` is drawn inside the `window`. Actually first the previous `canvas` is deleted and only then the new one is inserted. At the end of the game session the `window` is destroyed.

[1] The map is printed with several additional parameters related to the present status of the game. We report only the `visited_nodes`, the `visited_edges` and the possible available directions from the `current_position`. Additionally, the `current_position`, the `active_entities` and the non active ones are highlighted respectively in gray, red and green. Also some labels regarding `current_life`, `current_stamina`, `counter_games` and `seed` are included.

In Figure 5.1 are reported some screens of the key stages of a game session (still using the main example).

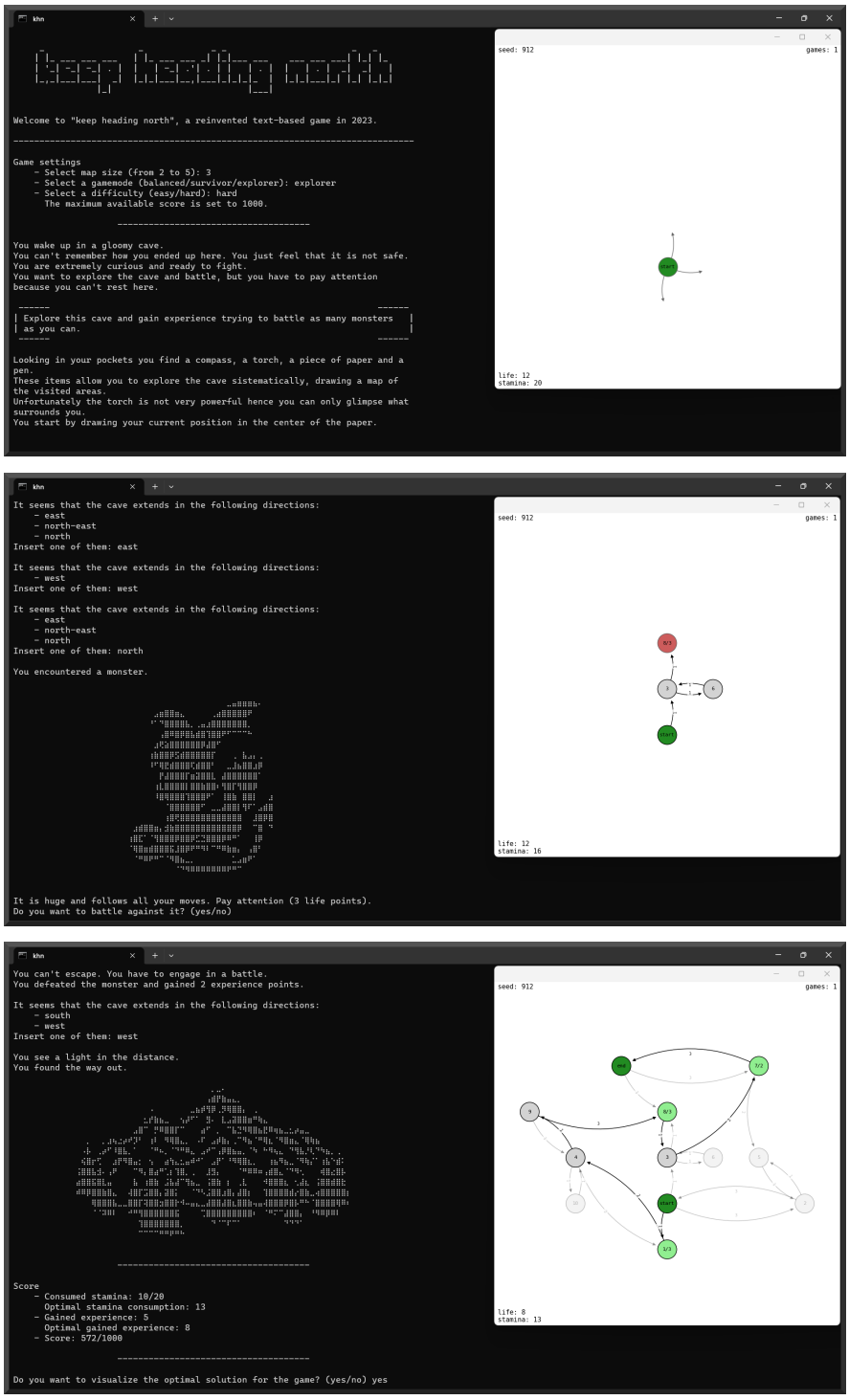

Figure 5.1: Screens of a game session: beginning, first monster encountered and optimal solution shown.

After the end of the game session `show_results` is called. Thanks to this function we compute the score obtained by the player from the various game parameters and the triple (`current_life`, `current_stamina`, `counter_games`) returned by `start_game`. The computation of the score is described in more details in Section 5.1.

The following two functions are intended to show how the game session should have been played. The first one just prints the `best_path` related to the couple (`best_stamina`, `best_life`) previously obtained by the function `get_parameters`. The second one is more complicated and will be discussed in Chapter 6: it prints a sequence of steps computed by a reinforcement learning algorithm that trained itself by playing several times the current game map.
The very last function called from the loop of `main` is `play_again` which simply lets the player keep on looping if he wishes.

Remark 5.2. All the player's input are managed through different dictionaries which allow the definition of some aliases. In this way the inputs result much more elastic and can be arbitraly extended. We report one of these dictionaries below.

```
aliases_gamemodes = {
    "balanced": "balanced", "b": "balanced", "1": "balanced", "": "balanced",
    "survivor": "survivor", "s": "survivor", "2": "survivor",
    "explorer": "explorer", "e": "explorer", "3": "explorer"
}
```

Listing 5.2: Portion of the file `aliases.py`.

5.1 Score

The final score associated to a winning game session is computed from 9 parameters: only 3 of them depend on the actual performance of the player, the remaining are setting parameters and outputs of `get_parameters`. We report the heading of the function `compute_score` in the Listing 5.3.

```
def compute_score(life, current_life, best_life, stamina, current_stamina,
 ↪ best_stamina, counter_games, gamemode, difficulty):
    """
    description:
        Compute the score of the current session.
    syntax:
        score = compute_score(life, current_life, best_life, stamina,
 ↪ current_stamina, best_stamina, counter_games, gamemode, difficulty)
    """
```

Listing 5.3: Heading of the function `compute_score`.

We introduced 6 functions which contribute to the creation of a fair result. We report in Figure 5.2 their plots and in Table 5.1 the related mathematical equations. We will refer to each of the first seven parameters using their initials.

As previously mentioned in Section 3.2 for the three different gamemodes the `experience` gained plays a different role. In `survivor` mode the `reward_experience` is not taken in account and similarly in `explorer` mode `reward_life` is ignored. What we did for the `balanced`

mode was just to consider the average between the two rewards. All the remaining bonuses weight equally on the final score for all the possible gamemodes:

$$\texttt{score} = \texttt{bonus\_life} + \texttt{reward\_stamina} + \texttt{bonus\_stamina} + \texttt{bonus\_games} + \\ + \begin{cases} \texttt{reward\_life} & \text{if } \texttt{gamemode} = \texttt{survivor} \\ \frac{\texttt{reward\_life} + \texttt{reward\_experience}}{2} & \text{if } \texttt{gamemode} = \texttt{balanced} \\ \texttt{reward\_experience} & \text{if } \texttt{gamemode} = \texttt{explorer} \end{cases}.$$

It is remarkable that, being sure to know the overall best performance for the current game, we decided to set all the maxima of our score functions to the values of `best_stamina` and `best_life`.

The `difficulty` parameter only influences the score at the very end. Indeed we just decided to reward the player which chooses to play in `hard` mode (i.e. with less initial `stamina` and `life`) by multiplying its final score by 2. Hence for an `easy` session the maximum obtainable score will be 500 whereas for an `hard` one will be 1000.
Thanks to the external loop of the function `start_game` (see Pseudocode 4) the player is allowed to try again to play on the same map also after he loses all its stamina or life. In order not to simplify too much the gameplay we capped the maximum number of restarts to 4 (`counter_games` < 4), as can be observed in the plot of the function `bonus_game` (Figure 5.2).

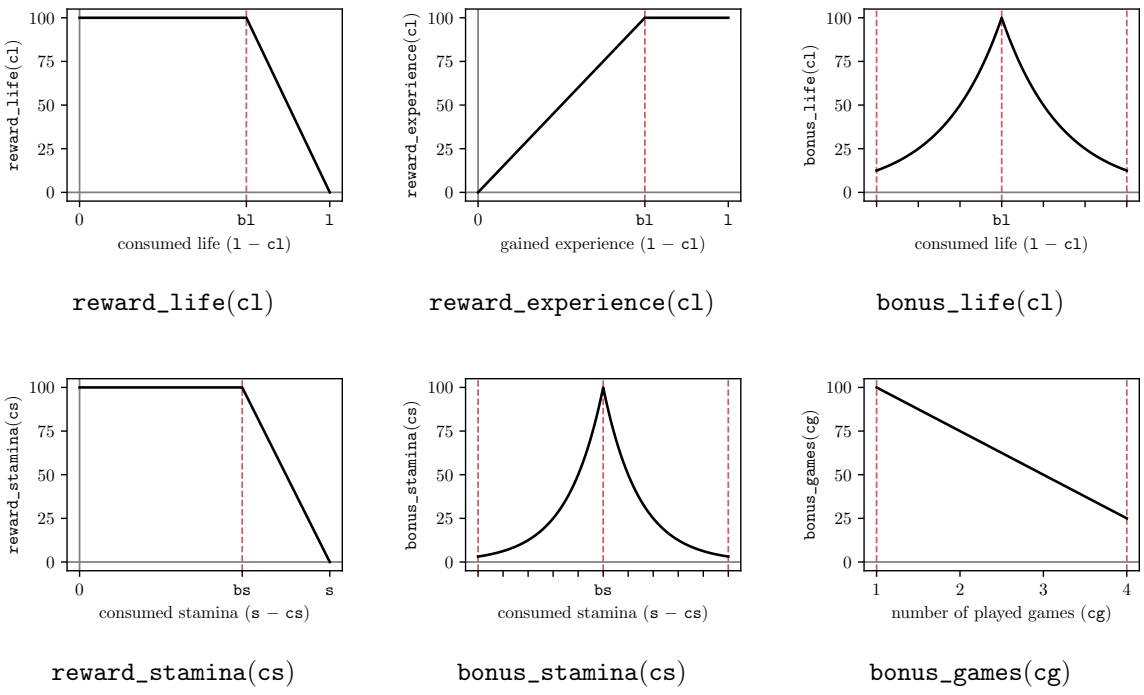

Figure 5.2: Plots of the 6 score functions.

function	equation
reward_life(cl)	$= \begin{cases} \min\left(\frac{\texttt{cl}}{\texttt{l}-\texttt{bl}} \cdot 100, 100\right) & \text{if } \texttt{bl} \neq 0 \\ 100 & \text{otherwise} \end{cases}$
reward_experience(cl)	$= \begin{cases} \min\left(\frac{\texttt{l}-\texttt{cl}}{\texttt{bl}} \cdot 100, 100\right) & \text{if } \texttt{bl} \neq 0 \\ 100 & \text{otherwise} \end{cases}$
bonus_life(cl)	$= \begin{cases} \frac{2^{4-\lvert\texttt{l}-\texttt{cl}-\texttt{bl}\rvert}}{2^4} \cdot 100 & \text{if } \lvert\texttt{l} - \texttt{cl} - \texttt{bl}\rvert \leq 3 \\ 0 & \text{otherwise} \end{cases}$
reward_stamina(cs)	$= \min\left(\frac{\texttt{cs}}{\texttt{s}-\texttt{bs}} \cdot 100, 100\right)$
bonus_stamina(cs)	$= \begin{cases} \frac{2^{6-\lvert\texttt{s}-\texttt{cs}-\texttt{bs}\rvert}}{2^6} \cdot 100 & \text{if } \lvert\texttt{s} - \texttt{cs} - \texttt{bs}\rvert \leq 5 \\ 0 & \text{otherwise} \end{cases}$
bonus_games(cg)	$= (5 - \texttt{cg}) \cdot 25$

Table 5.1: Mathematical equations of the 6 score functions.

6

Reinforcement learning player

The function `rl_solution` called by the `main` introduces the possible solutions which a simple Monte Carlo reinforcement learning algorithm would give to the last game session played. We will briefly recall the key concepts of reinforcement learning theory and then we will introduce the considered algorithm: on-policy first-visit Monte Carlo control.

6.1 Monte Carlo reinforcement learning methods

This theory is based on the idea of learning how to an agent can optimally accomplish a task based on its experience, requiring no prior knowledge of the environment's dynamics. Clearly we should not confuse the lack of an a priori knowledge to the absence of an underlying model. Indeed one of the core elements of a reinforcement learning (RL) model is the set of all possible transition probabilities from any state of the environment s_1 to any other state s_2 under an action a.

6.1.1 Markov decision process

A basic RL strategy is modeled as a Markov Decision Process (MDP) [3, p. 91], constituted of 5 parts:

- a set $\mathcal{S} = \{s_1, \dots, s_n\}$ of environment states;
- a family of sets $\mathcal{A}(s) = \left\{a_1, \dots, a_{m(s)}\right\}$ of actions available to the agent from any possible state $s \in \mathcal{S}$ $\left(\mathcal{A} = \bigcup_{s\in\mathcal{S}} \mathcal{A}(s)\right)$;
- a reward map $r : (\mathcal{S} \times \mathcal{A}) \times \mathcal{S} \to \mathbb{R}$ which associates a reward value $r \in \mathbb{R}$ to any couple initial state/action, arrival state $((s_1, a), s_2)$;
- a state-transition probability map $p : \mathcal{S} \times (\mathcal{S} \times \mathcal{A}) \to [0, 1]$ such that

  $$p(s_2 \,|\, s_1, a) \coloneqq p(s_2, s_1, a) = \mathbb{P}\left(\texttt{new\_state} = s_2 \,|\, \texttt{old\_state} = s_1, \texttt{action} = a\right),$$

  which inherits the notation from the conditional probability because of one of its main properties:

  $$\forall s_1 \in \mathcal{S}, a \in \mathcal{A}(s), \sum_{s_2\in\mathcal{S}} p\left(s_2 \,|\, s_1, a\right) = 1;$$

- a discount factor $\gamma \in (0, 1)$.

In our case this structure is, in some sense, simplified. Our state space $\mathcal{S}$ will be the set of vertices which constitutes the underlying graph of our map and, for each vertex v ($s \in \mathcal{S}$), $\mathcal{A}(s)$ is the set of edges starting from v.
The reward map will be better discussed later, but from now we can already say that one of its major components is constituted by the weights of the edges $(s_1, s_2) \in \mathcal{A}$.
Finally, the state-transition map we will consider is purely deterministic. Given $(s_2, s_1, a) \in \mathcal{S} \times \mathcal{S} \times \mathcal{A}$ we set

$$p(s_2 \,|\, s_1, a) = \begin{cases} 1 & \text{if } a \in \mathcal{A}(s) \text{ and } a = (s_1, s_2) \\ 0 & \text{otherwise} \end{cases}.$$

6.1.2 Policy and value functions

The latter building blocks we need to define our *RL* method are the policy π, the `state_value` and the `action_value` functions (v_π and q_π) [3, p. 97].
The policy is the probability measure which determines which action to perform when the agent is in state s,

$$\pi : \mathcal{A} \times \mathcal{S} \to [0,1], \text{ s.t. } \pi(a \,|\, s) \coloneqq \pi(a, s) = \mathbb{P}\left(\texttt{action} = a \,|\, \texttt{state} = s\right).$$

The agent has to select the move that maximizes the expected future reward. To this end two value functions can be defined: $v_\pi : \mathcal{S} \to \mathbb{R}$ and $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.
The value functions are repeatedly altered to more closely approximate the value functions for the current policy. Similarly, the policy is repeatedly improved with respect to the current value functions. These two subsequent changes work in parallel to create a moving target one for the other. Together they cause both policy and value functions to approach optimality.
The equations used in the implementation to update v_π and q_π after having generated an episode[1] $(s_{e_1}, a_{e_1}, r_{e_1}, \dots, s_{e_t}, a_{e_t}, r_{e_t})$, according to the policy $\pi^{(k-1)}$ (for $k = 1, 2, \dots$), are the following[2]:

$$g_{e_i}^{(k)} = \sum_{j=i}^{t} \gamma^j r_{e_j},$$

$$v_\pi^{(k)}(s_{e_i}) = \frac{1}{k} \sum_{j=1}^{k} g_{e_i}^{(j)}, \tag{6.1}$$

$$q_\pi^{(k)}(s_{e_i}, a_{e_i}) = r_{e_i} + \gamma \cdot v_\pi^{(k)}(s_{e_{i+1}}). \tag{6.2}$$

Furthermore, at each step k, the policy $\pi^{(k)}$ we consider is ε-greedy with respect to $q_\pi^{(k)}$, i.e. $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$

$$a_*^{(k)}(s) = \arg\max_{a \in \mathcal{A}(s)} q_\pi^{(k)}(s, a),$$

$$\pi^{(k)}(a \,|\, s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{\mathcal{A}(s)} & \text{if } a = a_*^{(k)}(s) \\ \frac{\varepsilon}{\mathcal{A}(s)} & \text{if } a \in \mathcal{A}(s) \setminus \{a_*^{(k)}(s)\} \end{cases}.$$

[1] An episode is a sequence of action choices which allows the agent to move through the states and gain rewards.

[2] The `returns` of each state s_{e_i} are denoted with g_{e_i}.

It is remarkable that the equations (6.1) and (6.2) are just the Bellman equations written in a simplified version taking in account the very basic state-transition map previously introduced (see [3, p. 59] and [2, p. 2–3]). Hence it applies the Bellman optimality Theorem [2, p. 3], reported below.

Theorem 6.1 (Bellman optimality)**.** Let an MDP $M = (\mathcal{S}, \mathcal{A}, r, p, \gamma)$ and a policy π. Then π is an optimal policy for M if and only if, $\forall\, s \in \mathcal{S}$ the value of $\pi(\cdot \,|\, s)$ is maximized in $\arg\max_{a \in \mathcal{A}(s)} q_\pi^{(k)}(s, a)$.

Hence the Theorem 6.1 ensures that our policy/value function update mechanism will eventually converge to an optimal policy.

Remark 6.1. It is important to note that if a couple $(s, a) \in \mathcal{S} \times \mathcal{A}$ does not appear in an episode, during the k-th iteration of the value functions update, we will not compute neither $v_\pi^{(k)}(s)$ nor $q_\pi^{(k)}(s, a)$. Hence, in order to perform correctly the policy update step, in these cases we will set $v_\pi^{(k)}(s) = v_\pi^{(k-1)}(s)$ and $q_\pi^{(k)}(s, a) = q_\pi^{(k-1)}(s, a)$.

The inital policy proposed is ε-greedy with respect to the reward function r. Indeed in our case the reward function can be seen as $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. It is because an action is an edge and so the only possible final state can be retrieved by the initial state s and the edge a itself. It follows

$$a_*^{(0)}(s) = \arg\max_{a \in \mathcal{A}(s)} r(s, a),$$

$$\pi^{(0)}(a \,|\, s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{\mathcal{A}(s)} & \text{if } a = a_*^{(0)}(s) \\ \frac{\varepsilon}{\mathcal{A}(s)} & \text{if } a \in \mathcal{A}(s) \setminus \{a_*^{(0)}(s)\} \end{cases}.$$

6.2 Rewards

In an initial phase the rewards have been thought so that the *RL* agent could reach the `end` of the map by consuming less stamina as possible. Then one could easily note that this strategy was just aiming to solve a shortest path problem.
In order to obtain this result we assigned the rewards as follows: $\forall\, s_1 \in \mathcal{S}, a = (s_1, s_2) \in \mathcal{A}(s_1)$

$$r(s_1, a) = -\texttt{weight}(a).$$

In addition we set $v_\pi^{(0)}(\texttt{end}) = 100$ so that during every iteration of the training the agent is pushed to move towards the end. This can be motivated by observing that, as we will see in the implementation of the *RL* algorithm (see Pseudocode 5), every episode ends only when the agent reaches the `end` or if it reaches a fixed `max_lenght`. Hence the state `end` is never added to an episode and so its `state_value` never changes: $\forall\, k \geq 0$, $v_\pi^{(k)}(\texttt{end}) = 100$.
Starting from this primitive strategy we extended the *RL* algorithm trying to include the possibility of finding optimal solutions for the various gamemodes. From the very beginning it was clear that the `survivor` mode was the easiest to implement because it was just a costrained shortest path where the agent should be forced to avoid the entities. On the other hand, a similar result for the `explorer` mode seemed accessible but nonetheless more challenging.

Ultimately, we decided not to try to approach the remaining `balanced` mode, since an optimal solution appears as a way less clear objective to aim to using the basic tools introduced so far.

6.2.1 Survivor mode

Attempting to make our agent avoid as many entities as possible, we modified the rewards in accordance with the following equations: $\forall\, s_1 \in \mathcal{S}, a = (s_1, s_2) \in \mathcal{A}(s_1)$

$$r_{\texttt{sur}}(s_1, a) = \begin{cases} -\texttt{weight}(a) - 20 \cdot \textbf{power}(s_2) & \text{if } s_2 \notin \textbf{entities}(\texttt{m}) \\ -\texttt{weight}(a) & \text{otherwise} \end{cases}.$$

Moreover, we set

$$\forall\, k \geq 0,\ v_\pi^{(k)}(\texttt{end}) = 100 \cdot \sum_{\texttt{e} \in \textbf{entities}(\texttt{m})} \textbf{power}(\texttt{e}) \tag{6.3}$$

to force the algorithm to head to the end of the map.

6.2.2 Explorer mode

Emulating what we just did for the `survivor` mode, we could just try to add positive rewards for the actions which conduct the agent to an entity. Unfortunately, this strategy results uneffective and often produces non-terminating instances. Indeed, the agent would be incouraged to loop over the entities without terminating the traversal, gaining as much reward as possible. The proposed solution includes a core modification of the MDP. The environment states are set to

$$\mathcal{S}_{\texttt{exp}} = \{(s, \texttt{active\_entities}) \,|\, s \in \mathcal{S},\ \texttt{active\_entities} \subseteq \textbf{entities}(\texttt{m})\}.$$

Similarly, $\forall\, (s_1, \texttt{active\_entities}) \in \mathcal{S}_{\texttt{exp}}$, the actions are modified to[3]

$$\begin{aligned} &\mathcal{A}_{\texttt{exp}}(s_1, \texttt{active\_entities}) = \\ &= \big\{\big((s_1, \texttt{active\_entities}), (s_2, \texttt{active\_entities} \setminus \{s_2\})\big) \,\big|\, (s_1, s_2) \in \mathcal{A}\big\}. \end{aligned}$$

At this point we can introduce the positive rewards associated to the entities, analogously with the `survivor` mode. This results in the following: $\forall\, (s_1, \texttt{active\_entities}) \in \mathcal{S}_{\texttt{exp}}$, $a = \big((s_1, \texttt{active\_entities}_1), (s_2, \texttt{active\_entities}_2)\big) \in \mathcal{A}_{\texttt{exp}}$

$$r_{\texttt{exp}}\big((s_1, \texttt{active\_entities}), a\big) = \begin{cases} -\texttt{weight}\big((s_1, s_2)\big) + 20 \cdot \textbf{power}(s_2) & \text{if } s_2 \notin \textbf{entities}(\texttt{m}) \\ -\texttt{weight}\big((s_1, s_2)\big) & \text{otherwise} \end{cases}.$$

One should notice that also the state-transition probability map necessitates a modification in accordance with $\mathcal{S}_{\texttt{exp}}$ and $\mathcal{A}_{\texttt{exp}}$.
This MDP update actually resolves the non-termination problems because now there is no advantage in traversing two times an entity state. Indeed, thanks to the construction of multiple states associated to the same vertex, we managed to introduce to the agent the concept of *defeated monster*.
As for the `survivor` mode, the `state_value` of the `end` is modified as in equation (6.3).

[3] Actually, whenever we write $A \setminus \{x\}$, we include the case in which $x \notin A$.

6.3 Implementation

Since we have completed the description the theoretical foundations of the *RL* strategy, we can now proceed by showing the pseudocode of the core algorithm (see Pseudocode 5, [3, p. 101]).

Pseudocode 5 On-policy first-visit MC control using ε-greedy policies.

1: **function** `on_policy_first_visit_mc_control(m, gamemode)`:
2: initialize parameters with `init_parameters(m, gamemode)`
3: **for** k **in** $1, \dots, 1000$ **do**
4: generate an episode following $\pi^{(k)} : (s_{e_1}, a_{e_1}, r_{e_1}, \dots, s_{e_t}, a_{e_t}, r_{e_t})$
5: $g = 0$
6: **for** i **in** $t, \dots, 1$ **do**
7: $g = \gamma \cdot g + r_{e_i}$
8: **if** e_i is the first occurrence of s_{e_i} in the episode **then**
9: **append** g to `returns`$[s_{e_i}]$
10: `state_value`$[s_{e_i}]$ = **average**(`returns`$[s_{e_i}]$)
11: `action_value`$[s_{e_i}, a_{e_i}] = r_{e_i} + \gamma \cdot$ `state_value`$[s_{e_i}]$
12: **for** s **in** $\mathcal{S}$ **do**
13: $a_* = \arg\max_{a \in \mathcal{A}(s)}$ `action_value`(s, a)
14: **update** $\pi^{(k)}$ to $\pi^{(k+1)}$ using a_*

The details related to the generations of the episodes (line 4 of the previous pseudocode) are always omitted by the textbooks because they are strictly related to the specific settings in which an algorithm is inserted. In our case this phase is pretty straighforward, we report it in Listing 6.1.

```
# loop until reaching the end or reaching max_iteration
max_iteration = 0
while state not in end and max_iteration < 1000:
    action = eps_greedy_policy(state, available_actions, proposed_actions[state], eps)
    episode.append((state, action))
    episode_states.append(state)
    state = action
    max_iteration += 1
```

Listing 6.1: Generation of an episode inside the on-policy first-visit MC control algorithm.

6.4 Considerations

We keep reporting the outcome of our code on the main example introduced in the previous sections (see Figures 6.1 and 6.2).

Although the *RL* algorithm seems to work very well with the majority of the maps, there exist some cases in which the proposed solution is non-optimal and rarely it is even non-terminating. This is completely reasonable, indeed the adopted strategy is pretty primitive and by construction not foresightful, therefore it could be definitely improved.

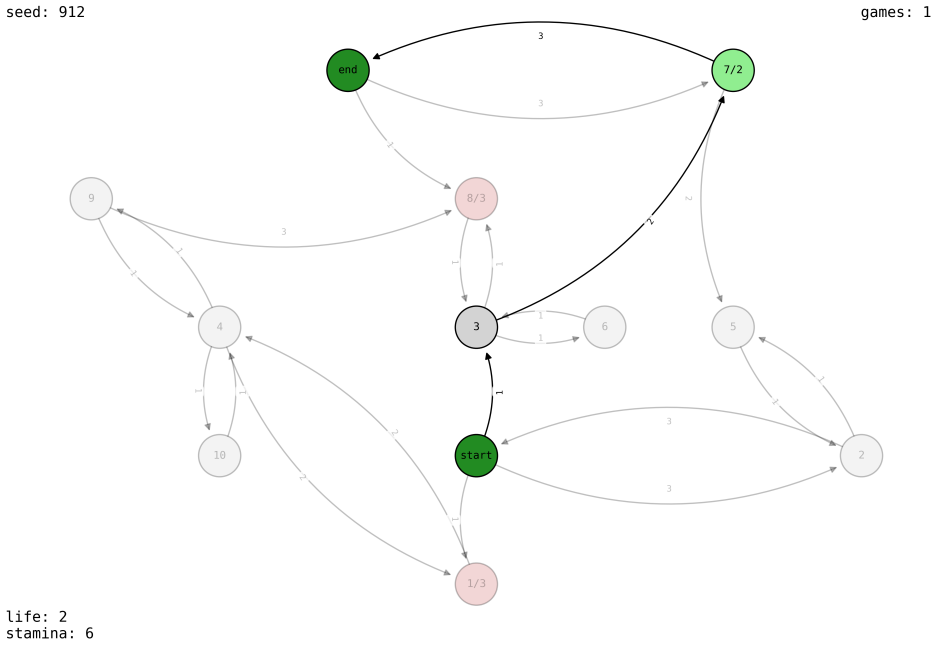

Figure 6.1: Main example: results of the *RL* algorithm for `survivor` mode.

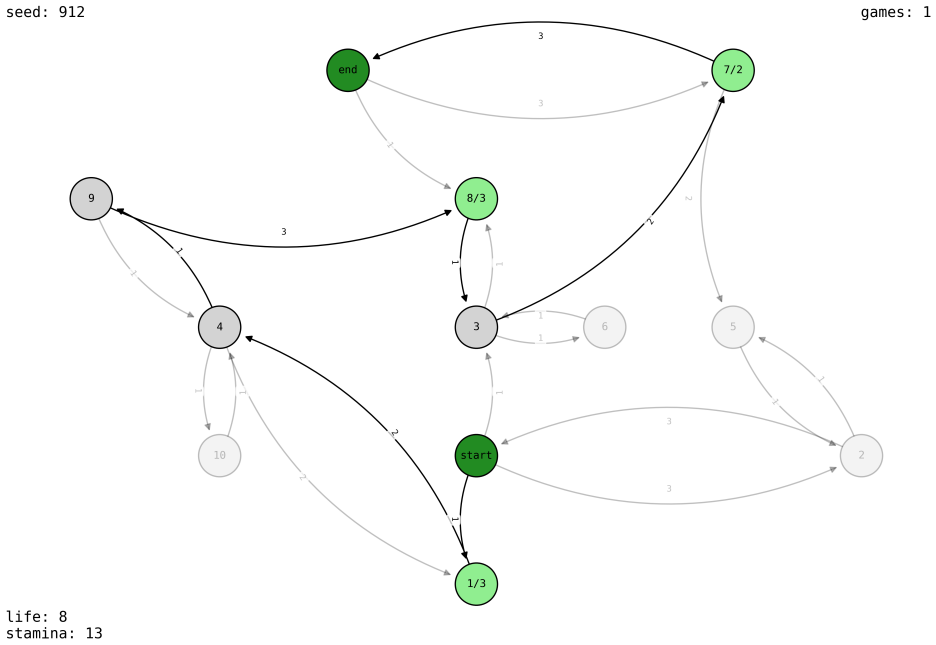

Figure 6.2: Main example: results of the *RL* algorithm for `explorer` mode.

7

Computational costs

In this last chapter we analyse the efficiency of our main algorithms. In particular we focus only on the two most relevant among them: `_generate` and `_compute_stamina_life`.

The complete codes, comprehensive of external functions, are reported in Appendix A, respectively in Listings A.1 and A.2. In the two following sections, first we will discuss the costs in time and after that we will briefly talk about the space consumption.
The only parameter against which is reasonable to make considerations is `bound`, so for the sake of brevity we denote it as `b` from now on.

7.1 Generation function

This algorithm, introduced in Pseudocode 1, is the easiest to study. The basic data structures employed in its development are dictionaries, lists (used to implement queues) and sparse matrices. Moreover we made use of the two classical visits of a graph, i.e. breadth first search (`bfs`) and depth first search (`dfs`).

Remark 7.1. We decided to implement the queues using `Python`'s lists so that we could exploit a native stucture. However it is important to note that the operation **pop** is very expensive in this case ($\mathcal{O}(n)$), hence we just decided to avoid popping by simply looping over the list, which is updated time by time.

The initialization of the `location` matrix costs $\mathcal{O}(1)$ thanks its sparse structure (line 56). The outer for loop runs until the queue `visited` is empty and each vertex can be inserted only once: this happens the first time we encounter it (line 120). Hence the number of iterations is equal to the number of vertices of the resulting graph. This corresponds to $\mathcal{O}\left(\mathtt{b}^2\right)$, indeed we have imposed the following constraint: $\mathbf{num\_vertices}(\mathtt{m}) \in [(2 \cdot \mathtt{b}+1)^2 \,//\, 5, (2 \cdot \mathtt{b}+1)^2 \,//\, 4]$ (see Remark 2.3 and condition 5 introduced in the Section 2.1, related to the generation of the graph). After this, another iteration immediately starts. In this case the analysis is easier, indeed it runs at most 3 times. The subsequent remarkable operation is the while loop at line 74, already analysed in Remark 2.1, which runs almost always less than 50 times. Inside this loop the only *expensive* operation is the call of the function `_find_vertex_from_coord` which costs $\mathcal{O}\left(\mathbf{num\_vertices}(\mathtt{m})\right) = \mathcal{O}\left(\mathtt{b}^2\right)$. Back to the previous loop we can observe that all the remaining operations cost $\mathcal{O}(1)$. Overall, the total contribution accumulated until now is bounded from above by $3 \cdot 50 \cdot (\mathtt{b}^4 + \mathcal{O}\left(\mathtt{b}^3\right))$.
Finally, the lines $139 - 149$ contain two non-negligible blocks: the first is related to the computation of the ending vertex through a `bfs` while the other concerns the condition 4 (Section

2.1) verified using a series of `dfs` visits. It is well known that both these algorithms cost $\mathcal{O}(n+m)$ with $n = \textbf{num\_vertices}(g)$ and $m = \textbf{num\_edges}(g)$, for any generic graph g (see [1]). In our case

$$\textbf{num\_vertices}(\mathtt{m}) = \mathcal{O}\left(\mathtt{b}^2\right) \text{ and } \textbf{num\_edges}(\mathtt{m}) = \mathcal{O}\left(8 \cdot \textbf{num\_vertices}(\mathtt{m})\right) = \mathcal{O}\left(\mathtt{b}^2\right),$$

hence both the functions cost $\mathcal{O}\left(\mathtt{b}^2\right)$. In view of these considerations we can state that the last chunk accounts for $\textbf{num\_vertices}(\mathtt{m}) \cdot \mathcal{O}\left(\mathtt{b}^2\right) = \mathcal{O}\left(\mathtt{b}^4\right)$ operations.
In conclusion the overall cost[1] of the function `_generate` is $\mathcal{O}\left(\mathtt{b}^4\right)$.

Remark 7.2. It is important to underline that all the three functions `check_adjacent`, `_insert_vertex` and `_insert_edge` require $\mathcal{O}(1)$ operations despite of they are non-trivial. It is due to the implementation of the graph using dictionaries which brings down the costs.

As for the cost in space apart from the dictionaries constituting the graph (which occupy $\mathcal{O}\left(\mathtt{b}^2\right)$), the heaviest variables are `visited` ($\mathcal{O}\left(\mathtt{b}^2\right)$) and `location`. The latter in the worst case becomes full and then accounts for $\mathcal{O}\left(\mathtt{b}^2\right)$ too.

7.2 Computation of stamina/life dictionary

As for the previous function, we first introduce the basic building blocks exploited during the computation of the dictionary `_stamina_life`, already discussed as Pseudocode 3: lists, tuples, dictionaries, sets and heaps. Basically `_compute_stamina_life` is an algorithm which aims to find optimal paths inside a structured graph (`map`), hence quite naturally we made use of the Dijkstra function.

The lines which preceed the first **for**, account for an $\textbf{num\_entities}(\mathtt{m}) = \mathcal{O}\left(\mathtt{b}^2\right)$ due to the creation of lists containing all the entities. From 62 to 68 we have the precomputation loop: this runs $\mathcal{O}\left(\mathtt{b}^4\right)$ times and during each iteration we perform $\mathcal{O}\left(\mathtt{b}^2 \log \mathtt{b}\right) + \mathcal{O}\left(2^{\mathtt{b}^2}\right)$ operations. The first addendum is a consequence of the application of Dijkstra implemented using binary heaps, which costs $\mathcal{O}((n+m)\log n)$ for a generic graph g with n nodes and m vertices (see [1]). The second one follows from the presence of the **powerset** function. Hence the whole block costs $\mathcal{O}\left(\mathtt{b}^4 \cdot 2^{\mathtt{b}^2}\right)$.
The main two nested loops together run $\sum_{k=0}^{n} \frac{n!}{(n-k)!} = \mathcal{O}\left(\left(\mathtt{b}^2\right)!\right)$ times, with $n = \textbf{num\_entities}(\mathtt{m})$. A rough analysis of their content leads approximatively to a total of $\mathtt{b}^2 + \mathtt{b}^2 + \mathtt{b}^2(\mathtt{b}^2 + \mathtt{b}^2 \log \mathtt{b} + 2^{\mathtt{b}^2}) = \mathcal{O}\left(\mathtt{b}^2 \cdot 2^{\mathtt{b}^2}\right)$ operations. Hence this second block would weight $\mathcal{O}\left(\mathtt{b}^2 \cdot 2^{\mathtt{b}^2} \cdot \left(\mathtt{b}^2\right)!\right)$. In any case the last part makes the precomputational contribution negligible.

This shallow analysis makes the algorithm appear excessively expensive and the usage of the `pairs` dictionary, which is built through powersets, an unnecessary burden. However, in practice, thanks to `pairs`, the number of operations inside the two nested loops is flattened. The calculation related to the average computational cost is omitted due to its complexity.
We report some data related to the amount of operations actually executed in Table 7.1.

[1] We are considering the final result without specifying the multiplicative constant, which actually is quite heavy (of the order of 1000).

The aim is to show how many executions of the function `_dijkstra` would be made without using the dictionary `pairs` (greedy case) and how many of them we perform (actual case). It is important to underline that in our case the heaviest operation in the loop starting at line 86 does not seem to be `_dijkstra` but the update of the dictionary `pairs` which costs exponentially. Nevertheless the update requires almost always way less operations than the worst case.
All these remarks contribute to a balance of the average computational cost and leads to an overall decent expense in terms of time.
We added the average number of entities for each value of the parameter `b` to the table because it corresponds to the actual value of almost all the $\mathcal{O}\left(\mathtt{b}^2\right)$ in the previous analysis (except for the ones related to Dijkstra).

`b`	2	3	4	5	6
entities	2	3	4.9	7	9
greedy case	9.4	39.6	1028.7	42844.6	3370489.8
actual case	4.5	14.8	125.9	683.4	4296.9

Table 7.1: Summary of average number of calls of the function `_dijkstra` in the main two nested loops.

Finally some considerations about the space consumption: the *larger* parameters introduced by `_compute_stamina_life` are the lists `entities`, `full_list`, `full_permutation`, `blacklist` which occupy $\mathcal{O}\left(\mathtt{b}^2\right)$ and the dictionaries `pairs`, `pairs_alias`. The latter are by far the biggest structures. In the worst case both of them could have a size of $\mathcal{O}\left(\mathtt{b}^2 \cdot \mathtt{b}^2 \cdot 2^{\mathtt{b}}\right)$. Furthermore the function `_dijkstra` also stores two non-negligible variables which are the set `seen` and the priority queue `heap`. Both of them account for an $\mathcal{O}\left(\mathtt{b}^2\right)$.

A

Codes

```
def _find_vertex_from_coord(self, coord):
    """
    description:
        Find the label of a vertex inside the graph starting from its
        coordinates (tuple).
        If the coordinates do not corresponds to any label it returns -1.
    syntax:
        v = m._find_vertex_from_coord(coord)
    """
    try:
        return
 ↪ list(self._coordinates.keys())[list(self._coordinates.values()).index(coord)]
    except:
        return -1

def _check_link(self, start, end, verbose = False):
    """
    description:
        Check whether there exists a path from start to end through a DFS visit.
        Verbose mode available.
    syntax:
        bool = m._check_link(start, end)
    """
    if (start == end):
        return True
    n = len(self._vertices)
    explored = [False] * n
    explored[start] = True
    stack = [start]
    while stack:
        u = stack.pop()
        for v in self._vertices[u]:
            if explored[v] is False:
                if v == end:
                    return True
                explored[v] = True
                stack.append(v)
    if verbose == True:
        print("error_check_link:", str(start) + "-" + str(end))
    return False

def _generate(self, bound, verbose = False):
    """
    description:
        Generate a map (vertices with their coordinates and edges) randomly.
        It takes as argument a parameter bound which is correlated to the size of the
 ↪ map.
        Verbose mode available.
    syntax:
```

```
48          bool = m._generate(bound)
49      """
50      # maximum size of the map / size of the matrix representing the game field
    ↪ (location)
51      #   bound * 4 + 1 = (bound * 2) * 2 + 1
52      # location: matrix of the game field
53      #   0: not visited nodes (new)
54      #   1: occupied nodes (vertex)
55      #   2: visited nodes (edge)
56      location = sparse.lil_matrix((bound * 4 + 1, bound * 4 + 1))
57      directions = ((0, 1), (1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1,
    ↪ 1))
58      max_step = 3
59      # max_vertex: number of possible vertices that could be inserted in the map
    ↪ divided by 4
60      max_vertex = (bound * 2 + 1) ** 2 // 4
61      # insert the start (0)
62      self._insert_vertex(0, (0, 0))
63      location[2 * bound, 2 * bound] = 1
64      # next_vertex: label of the next vertex to be inserted
65      next_vertex = 1
66      # visited: list of visited vertices
67      visited = [0]
68      for previous in visited:
69          # previous: label of the current position
70          prev_x, prev_y = self._coordinates[previous]
71          n_neighbours = random.randint(1, 3)
72          for _ in range(n_neighbours):
73              flag = False
74              while not(flag):
75                  # generate randomly direction and step
76                  direction = random.choice(directions)
77                  step = random.randint(1, max_step)
78                  # check of being inside the boundaries:
79                  #   if we are outside of the boundaries (vertically or horizontally)
    ↪ then continue
80                  if not(prev_x + direction[0] * step + bound in range(bound * 2 + 1))
    ↪ or not(prev_y + direction[1] * step + bound in range(bound * 2 + 1)):
81                      continue
82                  # try to find the arrival vertex (actually its label) inside the map
    ↪ (if it is not in the map it returns -1)
83                  vertex = self._find_vertex_from_coord((prev_x + direction[0] * step,
    ↪ prev_y + direction[1] * step))
84                  # backward edge:
85                  #   if the vertex is already in the map and the chosen edge has
    ↪ already been inserted
86                  #   then no additional control is required
87                  if vertex != -1 and self.check_adjacent(vertex, previous):
88                      break
89                  # overflow vertices:
90                  #   if we reached the maximum number of vertices in the map
    ↪ (max_vertex) and we are trying to add a new vertex
91                  #   then look for a new direction and step
92                  if next_vertex == max_vertex and vertex == -1:
93                      continue
94                  # check correctness edge:
95                  flag = True
96                  #   the flag is true if:
97                  #       the edge never passes on already taken nodes in the map (1 or
    ↪ 2 in location),
98                  #       the last half node is analyzed separately
99                  #       (check both integer and half nodes)
100                 for i in range(1, step):
101                     #   - half nodes
102                     flag = flag and (location[(prev_x + direction[0] * i + bound) * 2
```

```
    ↪ - direction[0], (prev_y + direction[1] * i + bound) * 2 - direction[1]] == 0)
103                     #   - integer nodes
104                     flag = flag and (location[(prev_x + direction[0] * i + bound) *
    ↪ 2, (prev_y + direction[1] * i + bound) * 2] == 0)
105                 #   - last half node
106                 flag = flag and (location[(prev_x + direction[0] * step + bound) * 2
    ↪ - direction[0], (prev_y + direction[1] * step + bound) * 2 - direction[1]] == 0)
107                 #   same check for the last node taking in account that it is allowed
    ↪ to end in an already taken node (if it is a vertex)
108                 #   (note: location[vertex] = 0/1, it cannot be 2)
109                 flag = flag and (location[(prev_x + direction[0] * step + bound) * 2,
    ↪ (prev_y + direction[1] * step + bound) * 2] in [0, 1])
110             # there is a probability of 0.5 to generate symmetric edges (back <= prob)
111             back = random.random()
112             prob = 0.5
113             # if we are adding a new vertex
114             #   then we insert both vertex and edge (also to the queue visited), in
    ↪ case we also add the backward edge
115             # else
116             #   if the edge haven't already been inserted then we add it (in case
    ↪ also the backward edge)
117             if vertex == -1:
118                 self._insert_vertex(next_vertex, (prev_x + direction[0] * step,
    ↪ prev_y + direction[1] * step))
119                 self._insert_edge((previous, next_vertex), step)
120                 visited.append(next_vertex)
121                 if back <= prob:
122                     self._insert_edge((next_vertex, previous), step)
123                 next_vertex += 1
124             else:
125                 if not(self.check_adjacent(previous, vertex)):
126                     self._insert_edge((previous, vertex), step)
127                 if back <= prob and not(self.check_adjacent(vertex, previous)):
128                     self._insert_edge((vertex, previous), step)
129             # update of the location matrix (following the rules at the beginning of
    ↪ the function)
130             for i in range(1, step):
131                 #   - half nodes crossed by an edge
132                 location[(prev_x + direction[0] * i + bound) * 2 - direction[0],
    ↪ (prev_y + direction[1] * i + bound) * 2 - direction[1]] = 2
133                 #   - nodes crossed by an edge
134                 location[(prev_x + direction[0] * i + bound) * 2, (prev_y +
    ↪ direction[1] * i + bound) * 2] = 2
135             #   - last half node crossed by an edge
136             location[(prev_x + direction[0] * step + bound) * 2 - direction[0],
    ↪ (prev_y + direction[1] * step + bound) * 2 - direction[1]] = 2
137             #   - last node occupied by a vertex
138             location[(prev_x + direction[0] * step + bound) * 2, (prev_y +
    ↪ direction[1] * step + bound) * 2] = 1
139     flag = True
140     # check number of vertices is in [(bound * 2 + 1) ** 2 // 5, (bound * 2 + 1) ** 2
    ↪ // 4]
141     if next_vertex < (bound * 2 + 1) ** 2 // 5:
142         if verbose == True:
143             print("error_generate:", next_vertex, "<", (bound * 2 + 1) ** 2 // 5)
144         return False
145     self._compute_end()
146     # check that it is possible to arrive to the end from any position
147     for u in self._vertices.keys():
148         flag = flag and self._check_link(u, self._end, verbose)
149     return flag
150
151 def _compute_end(self, start = 0):
152     """
153     description:
```

```
        Compute the end of the map through a BFS visit.
        It chooses the last vertex of the last layer of the BFS tree.
    syntax:
        m._compute_end()
    """
    n = len(self._vertices)
    explored = [False] * n
    explored[start] = True
    queue = [start]
    for u in queue:
        for v in self._vertices[u]:
            if explored[v] is False:
                explored[v] = True
                queue.append(v)
    self._end = u
```

Listing A.1: Source code related to the function `_generate`.

```
def _dijkstra(self, start, end, blacklist = []):
    """
    description:
        Dijkstra algorithm implemented with heaps. It is also possible to add a
        blacklist containing the vertices that we want to avoid
        (note that neither start or end should be blacklisted).
    syntax:
        distance, path = m._dijkstra(start, end)
    """
    n = len(self._vertices)
    distances = np.ones(n) * np.inf
    heap = [(0, start, ())]
    # seen = set() # normal dijkstra
    seen = set(blacklist) # blacklisted vertices are avoided
    while heap:
        dist, node, path = heappop(heap)
        if node not in seen:
            seen.add(node)
            path = path + (node, )
            if node == end:
                return dist, list(path)
            for neighbour, weight in self._vertices[node].items():
                if neighbour not in seen:
                    new_dist = dist + weight
                    if new_dist < distances[neighbour]:
                        distances[neighbour] = new_dist
                        heappush(heap, (new_dist, neighbour, path))
    # if it is impossible to find a path from start to end
    return np.inf, []

def _compute_stamina_life(self, verbose = False):
    """
    description:
        Create a dictionary that stores, for each possible life cost,
        the best stamina cost and its associated path.
    syntax:
        m._compute_stamina_life()
    """
    # entities: list of labels of nodes where the entities are located
    # full_list: entities + start + end
    entities = list(self._entities.keys())
    full_list = entities + [0, self._end]
    # pairs:
    #   - dictionary containing the shortest path which avoids the nodes in the tuple
 ↪ blacklist
    #       and its cost (tuple (cost, shortest_path)) for all couples in full_list;
```

```
    #   - the keys are triples of type tuple (node_1, node_2, blacklist);
    #   - we precompute pairs[(node_1, node_2, ())] for all couples (node_1, node_2);
    # pairs_alias:
    #   - dictionary which associates a triple (node_1, node_2, blacklist) with an already
    #       precomputed touple (cost, shortest_path);
    #   - the association is made following this assertion:
    #       if we merge the blackist list and a generic subset of the entities which are not blacklisted nor in the path
    #       then we obtain a new blacklist which still leads to the same shortest path
    # (note: actually some couples are excluded from pairs because they are not useful:
    #   - from any node to itself
    #   - from end to any node
    #   - from any node to start )
    pairs = {}
    pairs_alias = {}
    # verbose mode
    if verbose: n_precomputations, n_computations, n_greedy = 0, 0, 0
    for node_1 in full_list:
        for node_2 in full_list:
            if (node_1 != node_2) and (node_1 != self._end) and (node_2 != 0):
                if verbose: n_precomputations += 1
                pairs[(node_1, node_2, ())] = self._dijkstra(node_1, node_2)
                for alias in powerset(set(entities) - set(pairs[(node_1, node_2, ())][1])):
                    pairs_alias[(node_1, node_2, alias)] = (node_1, node_2, ())
    for k in range(len(entities) + 1):
        for permutation in itertools.permutations(entities, k):
            # permutation: ordered subset of the entities' set.
            #   we are looking for the shortest path that crosses these and only these entities, with order,
            #   starting from start and ending in end
            # flag: if the path we are considering is possible than True
            #   else False
            flag = True
            # life/stamina: variable containing the life/stamina consumed while traversing the path
            life = 0
            stamina = 0
            # path: list containing the shortest path associated with the current permutation
            path = []
            full_permutation = [0] + list(permutation) + [self._end]
            # at each step we look for the shortest path between node_1 and node_2 such that does not cross
            #   the entities which are still "active" (stored in the blacklist variable)
            blacklist = entities.copy() + [self._end]
            for i in range(len(full_permutation) - 1):
                node_1 = full_permutation[i]
                node_2 = full_permutation[i + 1]
                blacklist.remove(node_2)
                # if we haven't got the shortest path for the triple (node_1, node_2, blacklist) we compute it and store it
                #   (we also update pairs_alias)
                if verbose: n_greedy += 1
                if (node_1, node_2, tuple(blacklist)) not in pairs_alias:
                    if verbose: n_computations += 1
                    pairs[(node_1, node_2, tuple(blacklist))] = self._dijkstra(node_1, node_2, blacklist)
                    current_cost, current_path = pairs[(node_1, node_2, tuple(blacklist))]
                    for alias in powerset(set(entities) - set(pairs[(node_1, node_2, tuple(blacklist))][1]) - set(blacklist)):
```

```
98                          pairs_alias[(node_1, node_2, tuple(list(alias) + blacklist))]
↪ = (node_1, node_2, tuple(blacklist))
99                  # else we use the stored one
100                 else:
101                     current_cost, current_path = pairs[pairs_alias[(node_1, node_2,
↪ tuple(blacklist))]]
102                 if np.isfinite(current_cost):
103                     if i != 0:
104                         life += self._entities[node_1]
105                         path += current_path[1:]
106                     else:
107                         path += current_path
108                     stamina += current_cost
109                 else:
110                     flag = False
111                     break
112             if flag:
113                 # if the optimal path corresponding to the permutation costs less
↪ than the best path we found with the same life cost (life)
114                 #   we save in self._stamina_life[life] its stamina cost (stamina)
↪ and the path
115                 if (life not in self._stamina_life) or ((life in self._stamina_life)
↪ and (stamina < self._stamina_life[life][0])):
116                     self._stamina_life[life] = (stamina, path)
117     if verbose: return n_precomputations, n_computations, n_greedy
118     else: return
```

Listing A.2: Source code related to the function `_compute_stamina_life`.

B

Run the game

Machine requirements. The software `keep heading north` has been tested only on PC's powered by `Windows 11 (21H2)` and requires screens with resolution 1920×1080.

Software requirements. The game is though to run on a virtual environment named `khn_env_3.11` built with `Python 3.11.4` (with `tkinter`), hence the scripts contained in the folder `keep_heading_north\code\run\` should be used only after the construction of the `venv`[1]. After moving in the main folder of the game, the commands to create `khn_env_3.11` from a command prompt are the following:

```
cd keep_heading_north
py -3.11 -m venv khn_env_3.11
.\khn_env_3.11\Scripts\activate
py -m pip install --upgrade pip
py -m pip install pywinauto
py -m pip install networkx
py -m pip install matplotlib
py -m pip install scipy
```

Final refinements. In order to obtain the final version of the game as it was originally intended the user has to create a shortcut of the batch file `run.bat` (located inside the folder `run`) which runs the game. After the creation of the the shortcut (`.lnk`) one should open its properties and execute the following modification:

- rename the shortcut as preferred (e.g. `khn`);
- set the *Run* option to *Maximized*;
- change its icon inserting `keep_heading_north\code\source\images\khn.ico`.

Finally use the shortcut to run the game.

Alternative execution. It has been implemented a version of the game with simplified graphics which can be run in case the machine requirements cannot be both satisfied. It is sufficient to install `Python 3.11.4` (with `tkinter`) with the three packages `networkx`, `matplotlib` and `scipy`, then any user can open the Jupyter Notebook `keep_heading_north\code\source\main_notebook.ipynb` and run the code[2].

[1] The folder `keep_heading_north` corresponds to the main directory of the project.

[2] The `ipynb` packages are required. This method has been tested using `VS Code` and performs optimally by setting `"notebook.output.scrolling": true`.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] Andrew Ng and Stuart Russell. Algorithms for inverse reinforcement learning. *ICML '00 Proceedings of the Seventeenth International Conference on Machine Learning*, 05 2000.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.