Searching and Learning in Pac-Man

Albert Graells and Dave de Jonge

February 4, 2010

1 Introduction

In this Artificial Intelligence project we are asked to implement some intelligence on the player of a well-known computer game: Pac-Man.

We will start with a deterministic approach, considering the game without ghosts and trying to find methods that make Pac-Man eat all the dots. Firstly, we will desig a simple greedy solution that makes Pac-Man move all the time towards the nearest dot until there are remaining ones. After that, we will consider a harder problem: finding the shortest path that makes Pac-Man eat all the dots. Both problems will be solved using a well-known search algorithm: A^*. We will concentrate on the analysis and performance of the solution for the second problem, as the first one can be seen as a subproblem of it.

After solving this first problem, we will introduce randomness into the game, placing ghosts in the board. In this case, our goal is to *teach* Pac-Man how to move to avoid collisions with the ghosts while eating the maximum number of dots. We want this process to be done automatically, so we will give some positive reward to our Pac-Man agent each time it does a good movement and some negative one when it does a bad one. This procedure is known as reinforcement learning, and we will apply a specific technique called *Q-learning*. Finally, we will try several different rewards in order to appreciate changes in the behaviour of Pac-Man: aggressive, conservative...

To test our solutions, we have used the provided Pac-Man framework, implemented in Java, modified in such a way that we can use any programming language with it; more concretely, we have used C++. For all the tests present in this document, we have used a 2,4 GHz computer with 4GB of RAM.

2 Pac-Man searches shortest paths

The first approach for eating all the dots in the Pac-Man game is the following one: at each step, Pac-Man will look for the nearest not eaten dot and will go for it. This greedy strategy is, of course, not optimal, but it is a good starting point for trying to solve the problem of eating all the dots with the minimum possible number of movements.

This simple algorithm is shown in 1:

Algorithm 1 Eat all dots, greedy algorithm

```
EatAllDots($game$)
1 while MissingDots($game$) $\neq 0$
2        do $path \leftarrow$ GetPathToNearestDot($game$)
3           $game \leftarrow$ Advance($path, game$)
```

The procedure GetPathToNearestDot(*game*) can be implemented with an A^* algorithm. Any straight-forward implementation of this algorithm [2] can solve this problem in little amount of time, so we are not going to explain the details of our implementation. However, we will say some things about the possible heuristics used.

Three different heuristics can be used for this task:

- Zero heuristic: it is just a simple BFS. Of course it will have to explore some extra nodes, but it is easy to code.
- Manhattan heuristic: at each iteration, the Manhattan distance from a cell of the board to the nearest cell with dot is computed and this value is used as heuristic. Of course the number of nodes explored is lower than with the previous heuristic, as this one is more informed; however its execution time does not change so much because in each A^* we are forced to compute the minimum Manhattan distance from each cell to each dot and this computation takes more time than the search of the path.
- Shortest distance heuristic: at each iteration, the exact distance from a cell of the board to the nearest cell with dot is computed. In fact, these values can be pre-computed at the begining of the program using a Floyd-Warshall algorithm or one Dijsktra algorithm for each node [1]. With this heuristic, we don't explore any innecessary node but, again, there is no real improvement of the execution time because the time used in this precomputation is similar to the one used to solve the problem using a simple BFS.

The computer time used to solve the task is minimum: less than $100ms$ in all the cases.

3 Pac-Man learns to play the complete game

After eating all the dots in a Pac-Man game without ghosts with a greedy strategy, we want to solve it again but with the minimum number of movements possible. This problem is much harder than the previous one as the search space is huge, so a careful implementation is needed.

If we want to be successful in this task we have to find good ways to deal with the number of states explored, the exploration speed and the memory needed. These three concepts are clearly related to each other and it will not be possible to select the best method for one of them without causing problems with the other two, so the selection of the best overall algorithm cannot be divided into three diferent parts.

An outline of the main algorithm used to solve this task, based on A^*, is presented in section 3.2, but before, the representation of the state used is explained in 3.1 in order to show the real dimension and difficulty of the problem we are facing. The pruning heuristics used by A^* are in 3.4 and other implementation tricks such as memory management are detailed in 3.3.

3.1 State representation

The state used in this problem must contain, in a compacted way, the configuration of the full board; that is, Pac-Man's position and remaining dots.

A naive representation for the state would be having an integer to store the current position of the Pac-Man and an array indicating if a certain dot is eaten or not. Using a short integer (2 bytes) to store Pac-Man's position (we have 298 possibilities) and an array of 245 positions (1 byte in each position), then we would use 247 bytes for each state.

We can improve this last representation by using only $\lceil \log_2 298 \rceil = 9$ bits to store the position and another 245 bits for the dots. This means using 254 bits, compacted in a 32 byte structure. Of course the modification of this structure is harder than the previous one, but the memory used has improved a lot.

Another improvement we can do is collapsing all the nodes of a corridor into a single edge. With this representation, we will need not only to pass through each node with dots, but also through each edge that contains dots. In this way, from our initial board, we need to construct a graph that has as nodes the intersections of the board (see figure 1). Also we need to consider as nodes the initial position of Pac-Man and the positions with dots that have a non dot neighbour position, because these are possible positions where Pac-Man can end having eaten all dots (see nodes 11 and 12 of figure 1). It's important to see that with this approach, when Pac-Man is in a certain position, the possible movements it can do are:

- Moving from its position to a neighbouring one.

- Travelling through an edge.
- Travelling through an edge and returning to the initial position by changing the direccion of the movement just before reaching the neighbouring node.

Also we will need to consider that, when eating all the dots, it's possible that Pac-Man finishes not in a node but in a cell surrounding a node. This must be considered carefully in the implementation of the transition between states.

```
** ** ** ** ** 00 ** ** ** ** ** ** .. .. ** ** ** ** ** ** 01 ** ** ** ** **
** .. .. .. .. ** .. .. .. .. .. ** .. .. ** .. .. .. .. .. ** .. .. .. .. **
** .. .. .. .. ** .. .. .. .. .. ** .. .. ** .. .. .. .. .. ** .. .. .. .. **
** .. .. .. .. ** .. .. .. .. .. ** .. .. ** .. .. .. .. .. ** .. .. .. .. **
02 ** ** ** ** 03 ** ** 04 ** ** 05 ** ** 06 ** ** 07 ** ** 08 ** ** ** ** 09
** .. .. .. .. ** .. .. ** .. .. .. .. .. .. .. .. ** .. .. ** .. .. .. .. **
** .. .. .. .. ** .. .. ** .. .. .. .. .. .. .. .. ** .. .. ** .. .. .. .. **
** ** ** ** ** 10 .. .. ** ** ** 11 .. .. 12 ** ** ** .. .. 13 ** ** ** ** **
.. .. .. .. .. ** .. .. .. .. .. // .. .. // .. .. .. .. .. ** .. .. .. .. ..
.. .. .. .. .. ** .. .. .. .. .. // .. .. // .. .. .. .. .. ** .. .. .. .. ..
.. .. .. .. .. ** .. .. // // // 14 // // 15 // // // .. .. ** .. .. .. .. ..
.. .. .. .. .. ** .. .. // .. .. .. .. .. .. .. .. // .. .. ** .. .. .. .. ..
.. .. .. .. .. ** .. .. // .. .. .. .. .. .. .. .. // .. .. ** .. .. .. .. ..
// // // // // 16 // // 17 .. .. .. .. .. .. .. .. 18 // // 19 // // // // //
.. .. .. .. .. ** .. .. // .. .. .. .. .. .. .. .. // .. .. ** .. .. .. .. ..
.. .. .. .. .. ** .. .. // .. .. .. .. .. .. .. .. // .. .. ** .. .. .. .. ..
.. .. .. .. .. ** .. .. 20 // // // // // // // // 21 .. .. ** .. .. .. .. ..
.. .. .. .. .. ** .. .. // .. .. .. .. .. .. .. .. // .. .. ** .. .. .. .. ..
.. .. .. .. .. ** .. .. // .. .. .. .. .. .. .. .. // .. .. ** .. .. .. .. ..
** ** ** ** ** 22 ** ** 23 ** ** ** .. .. ** ** ** 24 ** ** 25 ** ** ** ** **
** .. .. .. .. ** .. .. .. .. .. ** .. .. ** .. .. .. .. .. ** .. .. .. .. **
** .. .. .. .. ** .. .. .. .. .. ** .. .. ** .. .. .. .. .. ** .. .. .. .. **
** ** ** .. .. 26 ** ** 27 ** ** 28 29 ** 30 ** ** 31 ** ** 32 .. .. ** ** **
.. .. ** .. .. ** .. .. ** .. .. .. .. .. .. .. .. ** .. .. ** .. .. ** .. ..
.. .. ** .. .. ** .. .. ** .. .. .. .. .. .. .. .. ** .. .. ** .. .. ** .. ..
** ** 33 ** ** ** .. .. ** ** ** ** .. .. ** ** ** ** .. .. ** ** ** 34 ** **
** .. .. .. .. .. .. .. .. .. .. ** .. .. ** .. .. .. .. .. .. .. .. .. .. **
** .. .. .. .. .. .. .. .. .. .. ** .. .. ** .. .. .. .. .. .. .. .. .. .. **
** ** ** ** ** ** ** ** ** ** ** 35 ** ** 36 ** ** ** ** ** ** ** ** ** ** **
```

Figure 1: Graph with collapsed edges.

With this representation, we have 37 nodes for the possible Pac-Man positions (6 bits), 30 nodes with dots and 43 edges with dots; that makes a total of 79 bits, so we can store them in a 10 byte structure.

A final refinement can be done to this approach. Let's consider for example node `00` and edge `00-03` of figure 1. There are two equivalent paths that consider this edge as a "turning back edge", one starting at `00` and returning to it and the other starting at `03` (see figure 2). In order to not consider both cases we will only consider the turning back option when the destination node has already been visited. In this way, if any of the edges of `00` are visited, we can be sure that also `00` is visited; on the other hand, if no edge is visited, obviously `00` has not been visited.

The consequence of this is that, if we store all the information about visited edges, we don't need to the information of `00` because we consider it "visited"

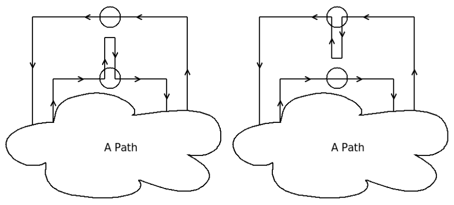

Figure 2: Two equivalent paths through a turning back edge.

when at least one of its edges is "visited" and "not visited" when all its surrounding edges are "not visited". The same technique can be applied to any of the nodes completely surrounded with edges with nodes. After this reducement, we have to store information of 7 nodes and of 43 edges, that together with the bits necessary to store Pac-Man's position, make a total of 56 bits, that can be stored in a 7 byte structure. This is the way we will represent a game state in order to solve it with minimum movements.

A summary of the different possibilities for state storing is showed in table 1. The size of the search space $Positions \cdot 2^{Dots}$ and the amount of allocatable states in 4GB is also included. It is quite obvious that we will need a clever algorithm to succeed in solving the game optimally, since the state space is huge.

Description	**Size (Bytes)**	**State space size**	**States in RAM**
Simple	247	$1.68 \cdot 10^{76}$	$1.62 \cdot 10^{7}$
Simple grouped	32	$1.68 \cdot 10^{76}$	$1.25 \cdot 10^{8}$
Compacted	10	$3.49 \cdot 10^{23}$	$4.00 \cdot 10^{8}$
Fully compacted	7	$4.17 \cdot 10^{16}$	$5.71 \cdot 10^{8}$

Table 1: State summary

3.2 General algorithm

For the solution of the Pac-Man shortest path problem we will also use A^* algorithm because it is complete (ie, it will always find a solution) and because it allows the use of an heuristic that will prevent us from exploring the whole state space of the problem.

One of the good points of A^* is that, if used with admissible and monotonic heuristics, the first solution that we get is the one with the lowest cost. On the other hand, we don't find any solution during the process apart from the optimal one; so, if A^* does not finish in a reasonable time (perhaps because the state space is huge), we don't have any solution, not even a suboptimal one. That's the reason why sometimes other methods are prefered. We have developed a method that can find non-optimal partial solutions while keeping some of the good properties of A^*.

3.2.1 Algorithm steps

The main idea of our algorithm is the following: let's imagine we are doing a conventional A^* with a monotonic heuristics $h(n)$. It may be possible that at a certain stage of its iteration process we generate a state with few remaining nodes to visit; let's call this state s, and the length of the best path from the starting node to reach it is $g(s)$. As this state has few remaining nodes, instead of using a typical heuristics function, we can compute the *exact* cost from this state to the final goal. That is, instead of computing $h(s)$, we can compute $h^*(s)$. It is obvious that we have found a solution that has an overall cost of $c_{up} = h^*(s) + g(s)$. Maybe the path we have found is not the optimal one, but its clear that it sets an upperbound of the cost of the optimal solution. From this, we can conclude that it is not necessary to include any node n in the opened list if $f(n) = g(n) + h(n) \geq c_{up}$ and, in this way, save some memory. We can continue exploring states and, perhaps, finding better values for c_{up}. In fact, it is possible that after some time the value we have for c_{up} is the optimal one, but the problem is that we don't know it. A visual explanation of these concepts is shown on figure 3.

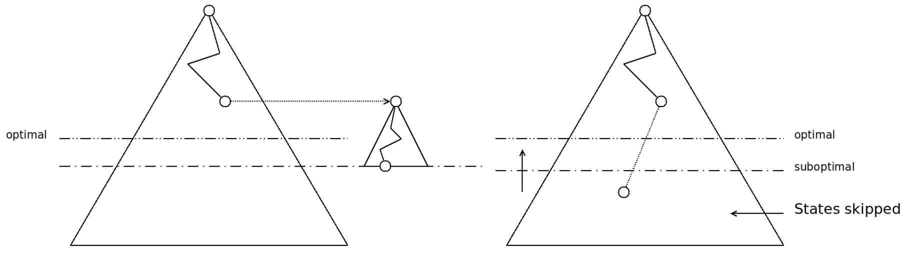

Figure 3: Algorithm proposed

Since the only way to know when we can stop looking for solutions is when we have $c_{up} = c_{optimal}$, we propose first to find the value of $c_{optimal}$ using a conventional A^*, optimized only for finding this value and not the path for getting it. Before that we can use our algorithm to find a value for c_{up} that can save some memory, and after that we have to use (again) our value for finding suboptimal solutions until we get one with cost equal to the optimal one.

Summarizing, the algorithm used consists of two different steps: finding the cost of the optimal solution and finding its path. It is clear that we are making this division in order to reduce the amount of memory used at the same time. Both steps are performed using A^*, one without saving parent-child relations but exploring more states, and the other one saving relations but exploring few states. In addition we are also using a third A^* to find the exact value for an heuristics when a state is not (computationally) far from the goal state. In this document, to refer to each one of the A^*'s used, we will use the following nomenclature: *exploring* A^* is the one used to find the cost of the optimal solution, *heuristics* A^* is the one used to calculate the exact value of the heuristic funcion and *path* A^* the one used to get the path that Pac-Man has to follow in order to eat all the dots.

In order to make the first step a bit faster, we can let the second algorithm run for some time so that we can get an upperbound of the cost of the solution and then use it to reduce the number of stored states in the first step, like it was shown in figure 3.

3.2.2 A* specific algorithm

An outline of the general A^* procedure, adapted for the Pac-Man problem, is in Algorithm 2.

Algorithm 2 A^* algorithm

```
GetOptimal($s_o$)
1  $L_o \leftarrow \{s_o\}$
2  $L_c \leftarrow \emptyset$
3  while $L_o \neq \emptyset$
4          do $s \leftarrow$ ExtractMin($L_o$)
5              if GoalState($s$) = true
6                 then return StateInfo($s$)
7              AddState($s, L_c$)
8              GenerateSuccessors($s, L_o, L_c$)

GenerateSuccessors($s, L_o, L_c$)
1  for each edge $e$ of GetPac-ManPosition($s$)
2          do AddStateIf(WalkEdge($s, e$), $L_o, L_c$)
3              if CanWalkBackEdge($s, e$) = true
4                 then AddStateIf(WalkBackEdge($s, e$), $L_o, L_c$)

AddStateIf($s, L_o, L_c$)
1  if Find($s, L_c$) = true
2      then return
3  if Find($s, L_o$) = true
4      then Update($s, L_o$)
5      else  AddState($s, L_o$)
```

For simplicity, we have not included the details of GetPac-ManPosition(s), WalkEdge(s, e), CanWalkBackEdge(s, e) and WalkBackEdge(s, e). The reason for this is that these procedures depend on which of the three A^* are we using or on the board. More precisely, they depend on the concept of state, s. Here we are considering as state a structure that contains the configuration of the board at that game step, the cost and the heuristic, and, perhaps, information about its parent state. In *path* A^* we will include the configuration of the previous game state, in *exploring* A^* we will not include anything (we don't want to use extra memory) and in *heuristics* A^* we will include some extra information regarding the heuristics of state's parent; the details of this information are explained in section 3.4.

Our final version of A^* does not follow exactly this version due to performance problems in UPDATE(s, L_o). The goal of this procedure is to search for s in the opened list L_o and update its cost if the new state generated has lower cost than the first one. The details of these performance problems are detailed in section 3.3 but are due to the fact that we want the opened list to be able to add states, update values and extract states with minimum cost plus heuristics in a reasonable time ($\mathcal{O}(1)$ or $\mathcal{O}(\log(N))$), while using the minimum amount of memory possible.

Algorithm 3 Alternative A^* algorithm

```
GetOptimal($s_o$)
 1  $L_o \leftarrow \{s_o\}$
 2  $L_c \leftarrow \emptyset$
 3  while $L_o \neq \emptyset$
 4        do $s \leftarrow$ ExtractMin($L_o$)
 5           if Find($s, L_c$) = true
 6              then goto 3
 7           if GoalState($s$) = true
 8              then return StateInfo($s$)
 9           AddState($s, L_c$)
10           GenerateSuccessors($s, L_o, L_c$)

GenerateSuccessors($s, L_o, L_c$)
1  for each edge $e$ of GetPac-ManPosition($s$)
2        do AddStateIf(WalkEdge($s, e$), $L_o, L_c$)
3           if CanWalkBackEdge($s, e$) = true
4              then AddStateIf(WalkBackEdge($s, e$), $L_o, L_c$)

AddStateIf($s, L_o, L_c$)
1  if Find($s, L_c$) = false
2     then AddState($s, L_o$)
```

In order to *relax* the requirements we need for the opened list, we used another version of A^* which partially uses a small ammount of extra memory, but reduces the complexity of L_c. This version, shown in Algorithm 3, adds each new generated state to the opened list (allowing repeated states in this list), but prevents exploring this repeated states by checking if they already are in the closed list before the exploration part.

3.3 A^* implementation

In this section the details regarding the data structures used to implement the opened and the closed list of A^* are explained.

3.3.1 Opened list

The requirements we have for our opened list structure are: items sorted increasingly by $g + h$, fast insertion, fast removal and low memory usage. The memory part is the one *really* important in our project, since our search space is huge.

For lots of problems we can us a simple priority queue, implemented using a heap, that gives us a logarithmic insertion and removal. The memory used by each item n is the size of the state structure, the size of an integer saving the value of $f(n)$ and the size of an integer saving the value of $h(n)$.

However, in this concrete problem, we can even use a better structure. As there are few values of $< f(n), g(n) >$, we can define a mapping between each pair of these values to an integer. This integer can be used as the index of an array that will contain, in each position, all the possible states with that combination of f and g. This makes insertion and removal $\mathcal{O}(1)$ and also we reduce the memory used as we only have to store the state structure. This structure can be seen in figure 4.

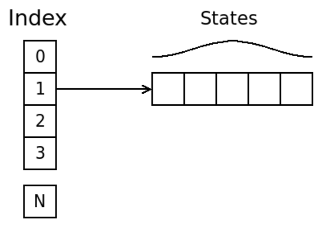

Figure 4: Opened list structure

3.3.2 Closed list

The requirements for our closed list structure are quite different from the ones of the opened list: we need the items stored in such a way that we can check quickly if an element is already in this set. Also we need fast insertion and, as always, low memory usage.

The typical data structure in a tiny problem would be a self balancing binary tree, like an AVL-Tree or a Red-Black-Tree [1]. These trees keep the data sorted even when inserting new elements, trying to keep the height of the tree low, with insertions and findings consuming logarithmic time. The main problem with these structures is that, for each node, we have to store the data, a pointer to the right node, a pointer to the left node and also some extra information about the level of the node. In our case, as our state representation only uses 56 bits, we would be using more memory with the pointers than with the data, so we need a better structure.

For this problem, we have used a hash structure similar to the one used for the opened list. In this case, a state s, consisting on 56 bits is split into two groups of bits, s_0 and s_1. The first group contains the higher 24 bits of the state, and the latter the remaining 32. s_0 can now we used as the index of an array that will contain, in each position, all the states with that value of s_0. Notice that in this case we only have to store s_1 and not the full value of s, that is, we are only using 4 bytes to store each of the states.

With this structure, we can find elements in linear time (locating the index of the array with s_0 in $\mathcal{O}(1)$ time and linear-searching into it in $\mathcal{O}(N)$ time (in fact this N is not very big as ideally it is the number of elements in the whole structure divided by 2^{24}). But there is still one last trick to increase the performance of this structure: as we are sure we will do more searches than insertions, we can keep the values with the same value of s_0 sorted and that will give us a linear insertion cost (as in the worst case we will have to move lots of elements), but a logarithmic search cost. We have empirically seen the total time of the algorithm decreases by a factor of 3 when using this. This structure can be seen in figure 5.

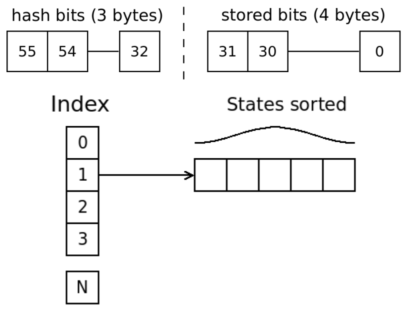

Figure 5: Closed list structure

3.4 Heuristics

Our goal is to find a way to calculate a lower bound for the length of the path that is as close to the real solution as possible. The play field can be considered as a weighted graph, in which the edges are the corridors of the maze and the vertices are the intersections of these corridors. The weight of any edge is its length. The goal is to eat all dots in the play field along the shortest possible path.

In order to explain the way we calculate the heuristic we need to get into some graph theory so first we will explain how to make an estimation of the shortest path length that traverses all edges, for a general weighted graph (we will use the terms *weight* and *length* interchangeably, so whenever we talk about the weight of an edge, we mean the length of this edge). In the last part of this section we explain how to apply this to Pac-Man.

3.4.1 Connecting Odd Vertices

Let's first consider the following simple situation: we have a weighted graph G with four vertices and six edges, as in Figure 6 and we want to find the shortest way to traverse all six edges (the actual weights are not displayed in this picture because their values are irrelevant for the following analysis).

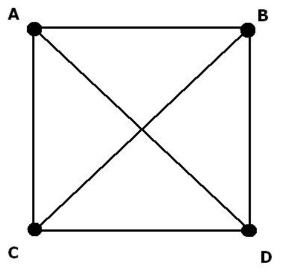

Figure 6: Graph G: a graph with four odd vertices.

Ideally we would like to traverse all of these edges exactly once. If this were possible then the length of this path would be exactly the total weight of the graph (that is: the sum of the weights of all the edges). If this is not possible than we can at least say that the length of the shortest path is always bigger than the weight of the graph, since every edge has to be traversed at least once. Therefore we can define the following admissible heuristic:

Definition 1. *The **total weight heuristic** h_{tw} is the sum of the weights of all the edges of the graph.*

In the case of Figure 6 it is certainly not possible to traverse all edges exactly once. To show that this is indeed impossible we take a look at the upper left vertex, vertex A (Figure 2). We see that it's a vertex where three edges come together, let's label these edges with 1, 2 and 3 respectively. Suppose that we try to walk over the graph without traversing any edge more than once, and at some moment we arrive at vertex A via edge 1. Then we can continue our walk along edge 2 or 3, let's choose 2. Of course, since all edges have to be traversed and we try to traverse them exactly once, at some moment we have to traverse edge 3 and we will get back to vertex A. But now we have no more possibilities to leave this vertex, because all of its edges have already been traversed. This means that either we have to double traverse one of the edges, or that vertex A is the endpoint of the path (or it was the starting point). But if we now look at the other vertices B, C and D we see that for these vertices we can make exactly the same analysis (all of them have an odd number of edges) and since of course we can have only one starting point and only one end point we have to conclude that it is impossible to traverse all edges without having to cross some of them at least twice.

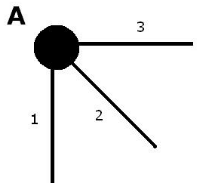

Figure 7: If A is not the starting point, nor the end point of the path, then one of its three edges has to be traversed twice.

Definition 2. *A vertex that has an even number of edges is called an* ***even vertex*** *and a vertex that has an odd number of vertices is called an* ***odd vertex****.*

Lemma 1. *For an odd vertex we are sure that one of its edges has to be traversed twice (except when this vertex is either the starting point or the endpoint of the path).*

So suppose that our starting point is C and our endpoint is D. We want to traverse all edges, but with minimal path length. That is: we want to traverse edges that have already been traversed before as least a possible. From the above lemma we see that both for vertex A and for vertex B we know that at least one of their edges has to be traversed twice. So the shortest solution would be to traverse the edge that is connected to both A and B (let's call this edge E_{AB}) twice.

Now we will look at the problem in a slightly different way: instead of demanding that edge E_{AB} is traversed twice we add another edge to the graph which is an exact copy of edge E_{AB} (this means it also connects vertices A and B together, and has the same weight). Let's call this edge E'_{AB} (see Figure 3.4.1). Then we can again assume that every edge is traversed only once, because instead of traversing E_{AB} twice, we can first traverse E_{AB} and then E'_{AB}. Also we can make another interesting observation: vertices A and B now have an even number of vertices. Therefore, the fact that we can traverse every edge once is in agreement with lemma 1. We will call the new graph that we have thus created G'.

Definition 3. *Edges like E'_{AB} that were added to the original graph to make it possible to traverse every edge once will be called* ***Artificial Edges****.*

Also another advantage of this presentation is that the total path length is now equal to the sum of all the weights of all the edges. This means that in order to find the shortest path that traverses all edges we simply have to add artificial edges in such a way that all odd vertices (except two) are made into even vertices.

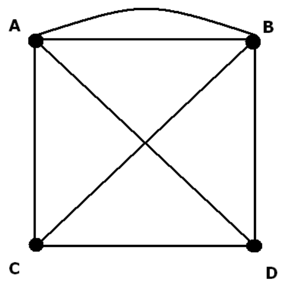

Figure 8: Graph G': equal to G, but with one artificial edge added to it. Vertices A and B are even vertices now and so we can walk along this graph traversing all edges exactly once.

Actually we can replace lemma 1 by a stronger one. Suppose now that G contained some more vertices between A and B. The graph G' would then look like Figure 9.

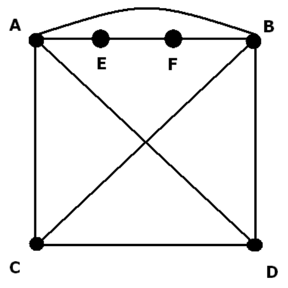

Figure 9: If there are even vertices on the path between A and B we have to add an artificial *path* rather than an edge.

We see that in this case nothing really changes to the analysis and the shortest path stays exactly the same. However, the artificial edge E'_{AB} that we introduced can now not really be seen anymore as the copy of the edge E_{AB}, because there simply is no edge E_{AB} anymore. This edge has become a *path* between A and B. Therefore we have the following lemma:

Lemma 2. *For any odd vertex (except the starting point and the endpoint) we are sure that one of its paths to another odd vertex has to be traversed twice.*

There are some small subtleties however. We see in lemma 2 that the starting point and the end point form exceptions. When our starting point is odd, then it might not be necessary to traverse any of its edges twice. On the other hand, if our starting point is even however, one can see that the opposite holds: we do have to traverse one of its edges twice. Therefore, in the rest of the analysis, when the starting point has an even number of edges we should consider it as an odd vertex and when it has an odd number of edges we should consider it as an even vertex. With this definition the starting point behaves exactly like other vertices, so we don't have to consider it an exception anymore. For the odd point we could make a similar analysis, but there is one problem: beforehand you don't know which vertex will be the end point of the path so it's impossible to tell whether it has an odd or an even number of edges. Therefore, we always have to take the end point into account as a special vertex. In practice this means that we always have to leave one odd vertex out when calculating the heuristic, because this vertex could be the end point. The vertex that we leave out is the one that results in the lowest value of the heuristic, so that we are sure the heuristic is admissible.

Lemma 2 proves that the following heuristic is admissible.

Definition 4. *The* ***odd-vertex-connection heuristic*** h_{ov} *is defined in the following way:*

1. *For a graph G find the cheapest way of adding artificial paths between odd vertices such that all these odd vertices (except one) become even. The thus obtained graph is denoted G'.*

2. *$h_{ov}(G)$ is now defined as the weight (the total length of all edges)of G'.*

Unfortunately it turns out that it is in general difficult to calculate the shortest way to pair a set of vertices. A problem in which you want to make pairs between vertices is called an *Assignment problem.* One of the most important algorithms to solve assignment problems is called the *Hungarian Algorithm*, which can solve them in $\mathcal{O}\left(N^3\right)$ time [4]. Directly applying this algorithm does not work, because the Hungarian algorithm assumes that the graph G is divided into two subsets S and T and that every vertex in S has to be matched with exactly one vertex in T. In our case however we have the set of odd vertices V and we want every vertex in V to be matched with a vertex also in V.

There are possible solutions to overcome this problem, but since this algorithm is quite complex and the time we have for the project is limited we decided to use another approach. A more practical solution that we have come up with is the following:

Definition 5. *The **suboptimal odd-vertex-connection heuristic** $h_{sov}(G)$ is defined as follows:*

1. *Start with the total weight heuristic $h_{tw}(G)$ of G.*
2. *Take an odd vertex and find the distance to the closest other odd vertex.*
3. *Divide this distance by two and add it to the heuristic.*
4. *Repeat steps 2 and 3 for all odd vertices.*
5. *Finally subtract again the highest value that you have added in the previous steps, to take care of the fact that we have to ignore the end point, as argued above.*

It is clear that the cost of this algorithm is $\mathcal{O}\left(V^2\right)$, where V is the number of vertices.

Notice that h_{sov} is always smaller than h_{ov} and therefore it is an admissible heuristic too.

3.4.2 Connecting Islands

The example graph G in the previous subsection was connected. In general however, a graph does not have to be connected and if it isn't we can put some more restrictions on the length of the shortest path.

Definition 6. *A maximally connected subgraph (that is: a subgraph to which we can not add any more edges without losing the property that it is connected) will be called an **island**.*

Lemma 3. *A path that traverses all edges (obviously) has to connect all islands with each other.*

From lemma 3 we see that just connecting odd vertices is not enough to solve the problem of traversing all edges once. To calculate the exact length of a path that traverses all edges we have to take into account the fact that one has to walk from one island to another. Therefore we can add artificial paths to the graph that connect the islands together.

Definition 7. *The **island-connecting heuristic** h_i is defined as follows:*

1. *For a non-connected graph G, add artificial paths in the cheapest possible way such that all islands are connected together. The thus obtained graph is called $\hat{G}$.*
2. *$h_i(G)$ is then defined as the weight of $\hat{G}$.*

To find these paths is easy: pick a vertex on one island and find the closest vertex on an another island. Add the path between the two vertices. Continue doing this until you have connected all islands. It is not hard to see that this is the shortest way to connect all islands.

The implementation of this algorithm is simple because it is just a modification of the the Kruskal algorithm [1], but starting with a set of connections already done. Its worst-case performance is $\mathcal{O}\left(E \cdot log(V)\right)$ where V is the total number of vertices and E the total number of edges. To deal with the set of connected vertices we can use a Disjoint-set data structure, that allows us to connect and check connections in $\mathcal{O}\left(\alpha(V)\right)$, where $\alpha(V)$ is the inverse of quickly-growing Ackermann function $A(V, V)$ [5].

3.4.3 A Combined Heuristic

In the previous two subsections we have seen that simply taking the weight of G is an underestimation of the length of the shortest path that covers G. We have seen that the real path length is longer because some paths between odd vertices have to be traversed twice, and also because we have to walk from one island to another. Therefore, we can improve our estimation of the length of the shortest path either by adding artificial paths between odd vertices or by adding artificial paths between islands. However, it would be even better if we did both: we would like to find the cheapest way of adding artificial edges such that all islands are connected and all odd vertices (except one) are connected.

Unfortunately this is where problems arise. Even if it were possible to find the cheapest way to join the odd vertices and also to find the cheapest way to join islands, doing both is not the same as finding the cheapest way to join odd vertices and islands *in the same time*.

We will show this with the following example. Suppose that we have two islands (displayed by circles) like in Figure 10 that both contain an odd vertex (vertices A and B). We will first try to join the islands together and then to join the odd vertices. Suppose that the shortest distance between the two islands is 2 and that the shortest distance between the two odd vertices is three. In Figure 10 we show what happens if you do this: by adding the second path, to connect the vertices, we also connect the two islands and therefore it wasn't necessary to add the first path. We could have just added the connection between the two odd vertices because it also connects the two islands. In the end we have increased the weight of the graph by 5, while we could have solved the problem with only an increase of 3. The heuristic is no longer admissible.

Therefore, instead of trying to find a way to solve both constraints in the same time we just solve them both separately and then take the maximum of both heuristics. So our final heuristic is:

$$h = \max\{h_{sov}, h_i\}$$

This heuristic is surely admissible, because h_{sov} and h_i both are. Notice however that is not monotonic, but that is not a problem because for the cost function we use the following definition:

$$f(n') = max\{f(n), g(n') + h(n')\}$$

where n is the parent of n', g is the path length. So that h effectively becomes monotonic.

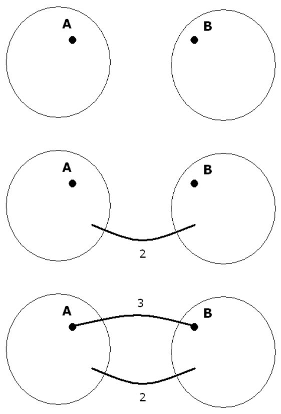

Figure 10: We have two islands, we first connect the two islands along there shortest path and then we connect the two odd vertices. But now the first path has become redundant.

3.4.4 Applying the Heuristic to Pac-Man

We have seen how, for a general graph, to make an estimation of the shortest path length that traverses all edges that is close to, but never bigger than, the real value. So how do we apply this to Pac-Man? The graph G that we have been talking about is the set of all corridors in the maze that contain dots. After all, this is exactly the part of the play field that we want to traverse. The intersections of these corridors then correspond to the vertices of the graph and the corridors themselves are the edges. The weights of the edges are simply the lengths of the edges (measured in steps). An important thing to notice is that the play field also contains corridors that are not filled with dots (either because they didn't contain dots at the beginning of the game, or because Pac-Man has eaten their dots in an earlier stage of the game). These corridors *do not form part of the graph G.*

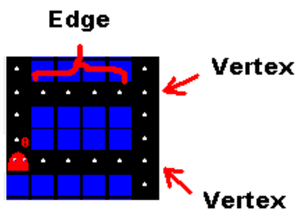

Figure 11: Edges and vertices in the play field of Pac-Man

Notice that the total weight heuristic is nothing more than the number of dots that are left on the play field. At the very beginning of the game this number is 245.

The island-connecting heuristic is the length of the corridors with dots, plus the number vertices with dots, plus the minimal cost for Pac-Man to walk between the islands (notice that a corridor between to islands cannot contain any dots, by the definition of an island), minus one. Someone could think that we are counting twice the dots that are on the vertices and that this would mean using a non admissible heuristic. In fact, we can safely do that because a vertex with N edges will be visited at least N times. The explanation of the *minus one* is that perhaps the last vertex is not really visited, if we had already visited it before. This heuristic starts at 257.

The suboptimal odd-vertex-connecting heuristic is immediately a lot higher than h_{tw}. At the beginning of the game it already has a value of 280. This suggests that h_{sov} is a lot better than h_i, however this is not really true. The fact that h_{sov} has a much higher value than h_i is due to the fact that at the beginning there is only one island (the whole play field is on big island), but there are many odd vertices, which can be quite far away from each other. As the game progresses more and more islands start to form so h_i becomes bigger and bigger, while in the same time the number of odd vertices becomes smaller. In fact, tests turn out that if we use h_{sov} without h_i we can't solve the problem because the computer runs out of memory. Using h_i individually on the contrary does give a result with acceptable use of memory. Therefore we can conclude that h_{sov} is only useful as an addition to h_i, but not on its own.

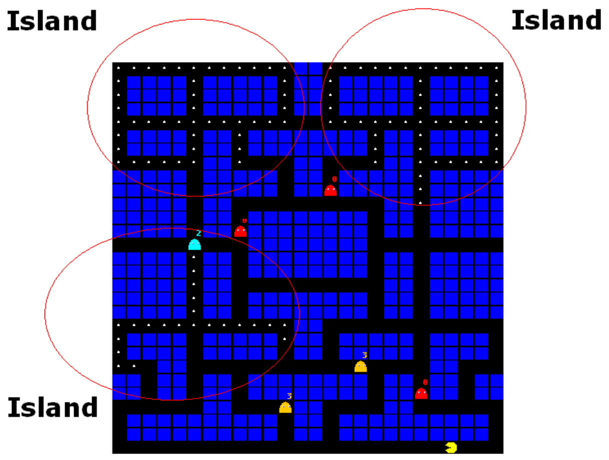

Figure 12: Islands in the play field of Pac-Man

3.5 Results

Our algorithm was able to find the optimal solution of the problem, which consists of 292 movements:

```
RRUUURRRRRRRRRRRDDDLLDDDRRDDDLLLLLLLLLLLUUURRRUUUL
LRRRRRDDDRRLLUUUUUUUUUUUUUUUUUUUUUDDDRRRRRUUULLLLR
RRRUUUULLLLLDDDUUULLLLLLDDDDRRRRRLLDDDLLLDDDLLLUUU
LLLUUULLLRRRRRRRRLLUUUULLLLLLDDDUUULLLLLDDDDRRRRLL
LLDDDRRRRRUUDDDDDDDDDDDDDDLLLLLDDDRRDDDRRRUUUUUURR
RRRRDDDLLLLLRRDDDRRRDDDRRLLLLLLLLLLLLLUUUR
```

To find out that 292 was the optimal cost the algorithm needed 50 minutes and, once it knew that 292 was the optimal cost, it needed less 15 seconds to find a solution with this cost, including its path.

To find the optimal cost, our A^* algorithm had to expand 121.116.056 nodes.

The total amount of memory used can be seen in figure 13. We can see the detail of the memory used in the open list and in the closed list. As expected, the memory in the closed list grows linearly as we explore new nodes in a constant rate.

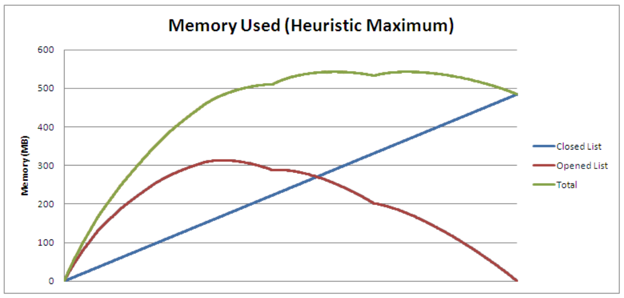

Figure 13: Memory used when using the maximum of both heuristics

We can see that finally we needed less than 600MB of memory; however we think that almost all the memory optimizations we did were really necessary.

We have also tried to solve the problem using each of the two heuristics individually. The heuristic that connects separate islands succeeded (see figure 14), but it needed more than two hours to find the optimal cost. The heuristic that deals with connecting odd vertices did not complete its execution, as it needed more than 4GB of memory (see 15). This is completely coherent, as one could think that a *good* way to eat all the dots in the Pac-Man game would be trying to keep all the missing dots together.

For the two heuristics giving an optimal solution, we registered the number of generated nodes for each possible value of f. We can see them in figure 16. It

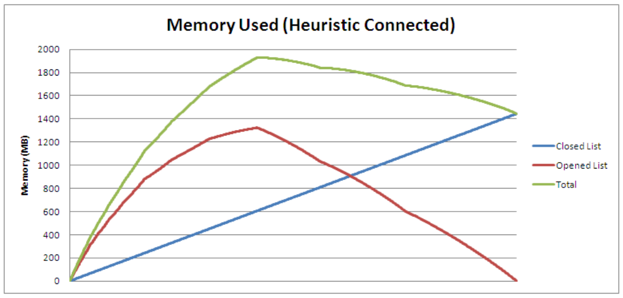

Figure 14: Memory used when using the island-connecting heuristic

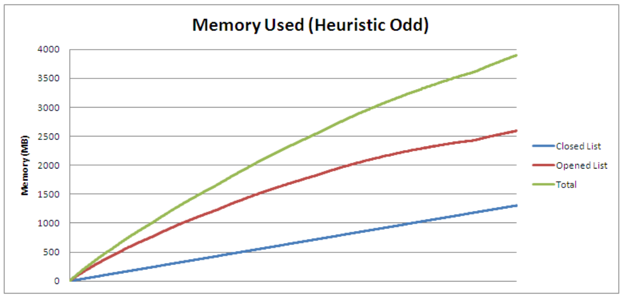

Figure 15: Memory used when using suboptimal-odd-vertex-connecting heuristic

may seem surprising that for the first values of the island-connecting heuristic we have very few nodes, but we have a lot for the max-heuristic. This happens because the latter heuristic is *less* monotonic than the first one. However, we can see that there is a lot of difference in the total number of generated nodes.

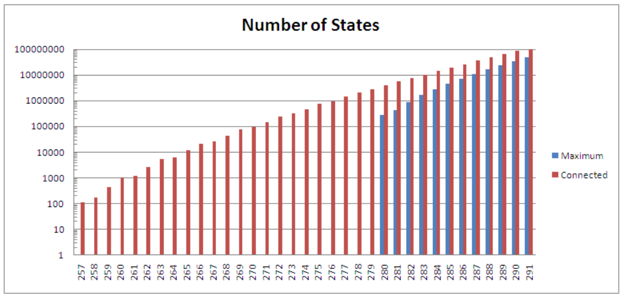

Figure 16: Nodes generated using each of the successful heuristics

3.6 Conclusions

We have solved the problem of eating all the dots in the game of Pac-Man in two different ways, both using an A^* algorithm. A simple implementation of the first one gives us a solution using a few ms of computer time, but this solution is not optimal. The second one computes the optimal solution, but it needs almost an hour to complete the whole task. It is infinitely harder finding the optimal solution to this problem than just finding a solution.

An A^* algorithm can be used for both tasks but is mainly used for the latter because, if we use an appropiate heuristic (admissible), the first solution found is the optimal one. When designing this algorithm, it is very important to use a good heuristic, otherwise we can fail in solving the problem, even when other aspects are highly optimized (as in the case of odd heuristic in figure 15). Other optimizations are also recommended, such as finding an appropiate state representation (in order to decrease the memory used to store a single state and also to reduce the state space size). Using the appropiate data structures for storing the generated and explored nodes is also necessary. However, none of these improvements will reduce the overall complexity of the problem, which remains exponential (unless we have a perfect heuristic).

It is unlikely that we could solve the problem if we had ghosts in the map: in the case that they are moving randomly it is not possible, as A^* only succeeds in deterministic cases, and in the case that their movements were predefined, the state space would grow a lot, as we could not use collapsed edges and we should keep the position of the ghosts in the state representation; we would also have to define different heuristics to deal with the positions (and future positions) of the ghosts. In this latter case we should use an algorithm that uses less memory, such as IDA^*, and wait for a long time (months?) until the solution is found. Perhaps for this problem it would be better to use a technique that gives us just a good solution, but not an optimal one.

4 Q-Learning

After learning how to solve optimally the game of Pac-Man without considering ghosts, it is time to introduce them and make Pac-Man succeed in the task of eating all the dots of the board without being killed. It is important that now we are facing a non deterministic problem.

To solve this task, we will use a Q-Learning algorithm adapted for our case. An outline of the algorithm can be found at [3].

The main problem with implementing Q-Learning for Pac-Man is that the number of possible states is huge. Therefore we need to find some kind of way to represent the real state as a simplified version.

4.1 Limited Window Representation

Our first attempt is to restrict the observable world to a small window of size 3x3 around Pac-Man. In other words: the learning algorithm can only observe the 8 cells directly around Pac-Man. Each of these cells can then be in one of four states: it can contain a ghost, a dot, a wall or nothing. This gives a world consisting of $4^8 \approx 66.000$ states (ignoring symmetries).

However, we encountered some problems with this representation. One problem is that often the state doesn't change at all after a move. This happens when Pac-Man is in a corridor where all dots already have been eaten. When this happens Pac-Man very easily gets stuck in repetitive moves. We could try to penalize repetitive moves, but this mainly seems to result in Pac-Man getting stuck in more complex repetitive patterns.

Another problem with this representation is that states often change in a very unpredictable way, because we don't have any information about the world more than one step away from us. Now in general uncertainty does not have to be a problem for Q-learning, in fact one of the strong points of Q-learning is that it can deal very well with randomness. The reason for this is that if you let the learning algorithm run for a while and we assume that the world is governed by a Markov decision process, the algorithm averages out all possible outcomes of every state-action pair. However, for our Limited Window Representation this doesn't work very well, because *in this representation the world is not a Markov decision process.*

Notice that in the beginning of the game there are a lot of dots in the play field. So if Pac-Man eats a dot, there is a very big chance that Pac-Man will find another dot in the next step. At the end of the game however, when there are only a few dots left, this is no longer the case. So the probability of finding a dot depends on the history of the game. Therefore, things that happen at the beginning of the game can have a very big influence on the rest of the game, which can not be averaged out by the law of big numbers, because as time passes, the game changes.

4.2 Relevant Information Representation

Another option that we have considered was, instead of limiting our view to local information, limiting our view to *important* information. That is: we consider only the locations of the ghosts, the location of the nearest dot and the location of Pac-Man. However, this is still way too much information to handle, since the play field consists of more than 250 possible positions and there are four ghosts. So this would take $250^6 \approx 2.4 \cdot 10^{14}$ possible states, which is huge. We could take some symmetries into account and simplify some things, but it is clear that with such big numbers this is not going to help much.

4.3 Fuzzy Distance

The state representation that we have used is the following: for each possible direction that Pac-Man can move in we check how far the nearest dot is when we take a step in that direction and we do the same for the nearest ghost. Then instead of storing the actual distances, we assign them a qualitative value in the range 1 to 3:

1. the ghost or dot is less then 2 steps away.
2. the ghost or dot is 2 to 9 steps away.
3. the ghost or dot is 10 or more steps away.

So for every direction we have three possible values for the distance to the nearest dot and three possible values for the nearest ghost. However, it is also possible that Pac-Man cannot walk in a certain direction because there is a wall, so we need another value to indicate that this is the case. Therefore we have $3 \cdot 3 + 1 = 10$ possible values for each direction. But we want to make one more refinement: when the nearest ghost and the nearest dot have the same value in a certain direction, we want to use an extra bit to indicate which one of the two is closer, therefore there are in total 13 possible values for each direction. The world then consists of $13^4 = 28.561$ possible states.

4.4 Symmetries

For whatever representation we use there are always some symmetries in the play field that we can exploit to decrease the size of the state space. In general you can always rotate the play field 90 degrees without really changing the situation. But at least for the representation that we have chosen there are even more symmetries. The only relevant information in our state representation is the actual set of values, but not the spatial orientation of these values. What we mean by this is that every permutation of the directions results in an equivalent state, so for example the following three states can all be considered equivalent:

	4	
5		3
	6	

	3	
5		4
	6	

	3	
6		5
	4	

The motivation for this is that Pac-Man should simply choose the direction which has the best value (closest dot, furthest ghost), while it doesn't make any difference which direction this is and which value the direction exactly opposite has. Actually, in some cases a permuted state might not exactly be equivalent, but we think that the difference is small enough to consider it equivalent in practice.

If a state consists of four different values, its equivalence class consists of $4! = 24$ equivalent states, so we can decrease the size of the world by a factor of approximately 24. This is not exact, because if a state has less than four different values its equivalence class is smaller. However, on the final implementation of our agent, we have not considered this simplification because the speed of convergence was already high enough.

4.5 Learning

In order to evaluate how well Pac-Man learns for different settings of parameters we have set up the following experiment. A test session is defined as a sequence of 250 games and in each game Pac-Man starts with 25 lives. During the whole session Pac-Man keeps its Q-table and frequency-table so it can continue learning even though it gets killed many times. Notice however, that there is a difference between playing 250 games with 25 lives and playing only one game with $250 \cdot 25$ lives. The difference is that in the first case, after every 25 kills, the play field is re-initialized (that is: all eaten dots are placed back in the play field) and the level is set back to 1. We chose this set-up because it sometimes happens that Pac-Man gets into a local minimum where it has eaten all dots in its neighborhood and doesn't look for new dots anymore. Because there are no more dots in its neighborhood it won't receive any rewards for eating dots anymore and therefore it will not learn to look for new dots. It gets stuck in a vicious circle and to break this circle we need that the play field gets re-initialized once in while.

To define the ratio between exploration and exploitation we used the following formula: for the first ten games Pac-Man moves 100% randomly. Then, after each game the probability of making a random move is decreased by a certain amount δ which we will call the *randomness decrease*. When the probability of making a random move becomes lower than 1% we set it exactly equal to 1%, so it will be 99% greedy for the rest of the session.

To evaluate how well Pac-Man learns we played with two parameters: the randomness decrease δ, and the reward for being eaten by a ghost ϵ. The reward for eating a dot was constantly set to 1. We tried five different settings of these parameters:

- strategy 1: $\delta = 5\%, \quad \epsilon = -1000$
- strategy 2: $\delta = 2\%, \quad \epsilon = -1000$
- strategy 3: $\delta = 100\%, \quad \epsilon = -1000$

- strategy 4: $\delta = 5\%$, $\epsilon = -10.000$
- strategy 5: $\delta = 5\%$, $\epsilon = -100$

4.6 Results and Conclusions

In the following figures we present the results of the Q-learning assignment. In Figure 17 we show the number of moves that Pac-Man makes during a game (averaged over the past ten games), as a function of the number of played games. This can be seen as a measure of how well Pac-Man learns to avoid ghosts. In this graph we can see clearly that the results go up much faster when δ is high. That is: the faster it becomes greedy, the faster its results go up. However, the final result for this strategy turns out not to be any better than for the other strategies. The only strategy that seems to be truly better than the others is strategy 4, which has a higher penalization for being eaten, than the other strategies. This result makes sense: the harder it is penalized for being eaten, the longer it will survive. However, it is not clear whether this result is significant.

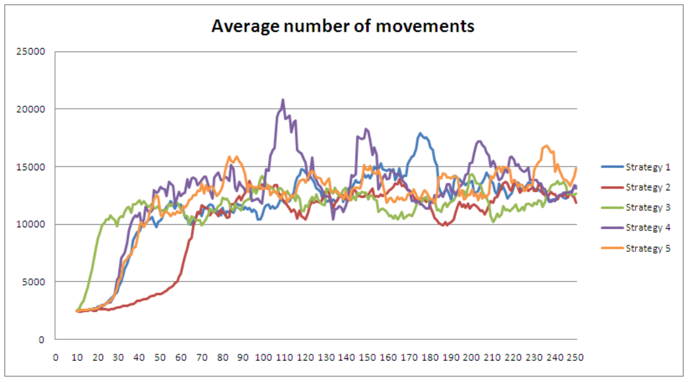

Figure 17: Sliding window average (window size: 10 games) of the number of moves made during a game.

In Figure 18 we see a sliding window average of the level to which Pac-Man reaches during a game. This can be seen as a measure of how well Pac-Man learns to eat dots. In this figure we see again that getting greedy faster, leads to good results faster, but this does not say anything about the final performance. What we also notice is, that in order to learn to eat dots, the strategy with the lowest value of ϵ gives best results. Again this makes sense. When Pac-Man is less afraid of ghosts it can focus more on eating dots, so it advances to a higher level. Of course, if we would make ϵ even lower it might happen that it learns almost nothing anymore from being eaten and therefore gets killed very often, which would prevent him from advancing to a high level.

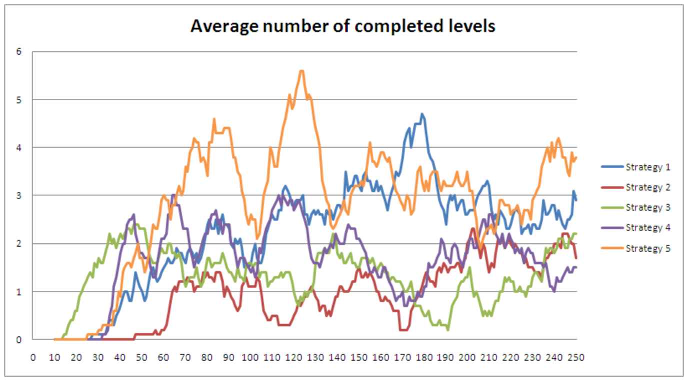

Figure 18: Sliding window average (window size: 10 games) of the level to which Pac-Man advances during a game.

Finally, to see if we could make any improvements by letting Pac-Man learn for a longer amount of time we have plotted the results in Figures 19 and 20 during a session of 1000 games. We see from these figures that learning a longer amount of time does not improve the results. Appearantly the Q-values have already converged after 250 games. If we had more time for the project it might have been a good idea to test directly how much the Q-values have converged.

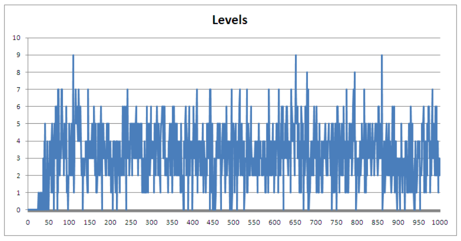

Figure 19: Levels after playing a larger number of games per session

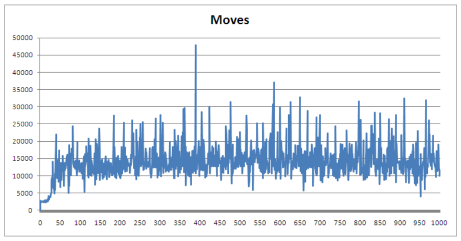

Figure 20: Moves after playing a larger number of games per session

References

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest (2001), *Introduction to algorithms (2nd Edition)* . MIT Press / McGraw-Hill.

[2] Stuart Russell, Peter Norvig (2002), *Artificial Intelligence: A Modern Approach (2nd Edition)* . Prentice Hall.

[3] Thomas Mitchell (1997), *Machine learning*. McGraw-Hill Higher Education.

[4] X-Ray, TopCoder Member (2008), *Assignment Problem and Hungarian Algorithm*. Algorithm Tutorials at `http://www.topcoder.com/tc`.

[5] Wikipedia (2010), *Disjoint-set data structure*.
`http://en.wikipedia.org/wiki/Disjoint-set_data_structure`