Reinforcement learning - Lab 1

Dimitri Diomaiuta - 30598109

University of Southampton

1 Fibonacci numbers

Fibonacci numbers are a mathematical sequence which values are determined by the recurrence relation 1 [4].

$$F_n = \begin{cases} 0 & \text{if } n = 0. \\ 1 & \text{if } n = 1. \\ F_{n-1} + F_{n-2} & \text{otherwise.} \end{cases} \tag{1}$$

The calculation of the fibonacci sequence shows the optimal substructure property. This means that the end result can be seen as a composition of solving subproblems of the same type using the same procedure. A plain recursion approach will generate, in the winding phase, a recursive tree of subproblems until the problem cannot be decomposed any further (the leaves of the tree). In the unwinding phase the generated solution of the subproblems are combined to solve a bigger subproblem. The solution is calculated when all the subproblems are solved and combined together. We can observe a recursion tree of the fifth fibonacci sequence number in figure 1. The recursion generates and traverses the

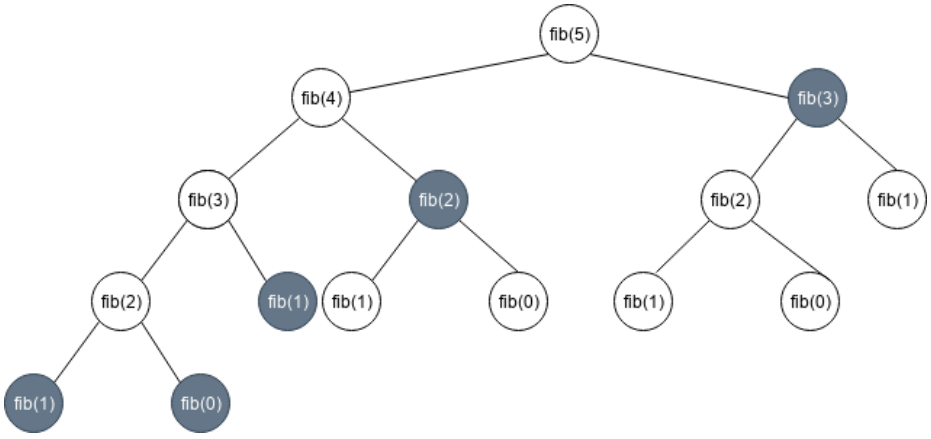

Fig. 1. Recursion tree of fibonacci sequence calculation

tree as a depth first search algorithm. We can observe from figure 1 that the

very same subproblems are generated and solved multiple times. We can observe that the subtrees that have a gray node as root have already been computed [3]. The dynamic programming paradigm solves exactly this kind of problem by exploiting memoization of the partial solutions [2]. When a subproblem is solved its solution is cached and reused once the same subproblem occurs. The dynamic programming optimization drastically improves the complexity time. The plain recursion approach runs in exponential time complexity, $O(2^n)$, while the dynamic programming method has a linear time complexity, $O(n)$ [3]. The space complexity of both the algorithms is linear, $O(n)$, since the tree is traversed in a depth first search like manner. The code of both methods can be seen in subsection 4.2 of appendix4.

2 Shortest path problem

The shortest path problem is another type of problem that has the optimal substructure property [5]. The Dijkstra's algorithm is one of the most used algorithms for the purpose of calculating the shortest path between a source and a target node. Splitting the shortest path problem in subproblems results in calculating the shortest paths between the source node and the nodes that stand in the way to reach the target node. Following the dynamic programming approach, Dijkstra algorithm uses memoization to store the solutions of the subproblems. In this section we extended the described algorithm to calculate the shortest paths for all the nodes in a graph. Each node is visited in an iterative manner and the subproblem solutions are stored in a matrix. We tested the program on a 100 nodes graph with uniform cost connections (path cost 1) with different probabilities of occurrence. Figures 2, 3 4 and 5 show the shortest pairwise distances distribution under different probability distributions. As we can observe from the histograms, increasing the connection probability decreases the average shortest path cost between each node. The code of the program can be seen in subsection 4.2 of appendix 4.

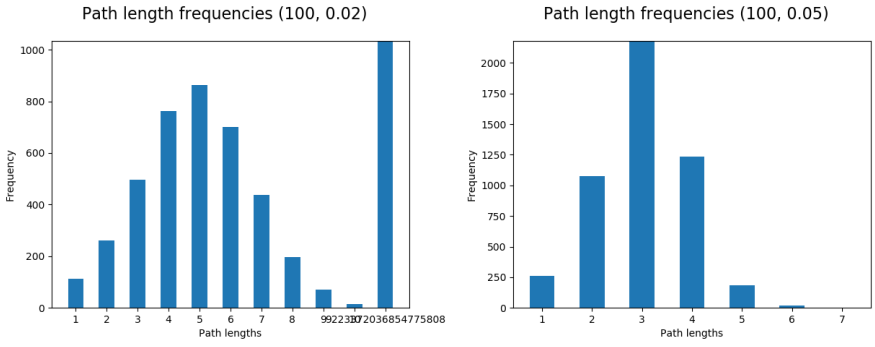

Fig. 2. Histogram of shortest pairwise distances for probability 0.02 and 0.05

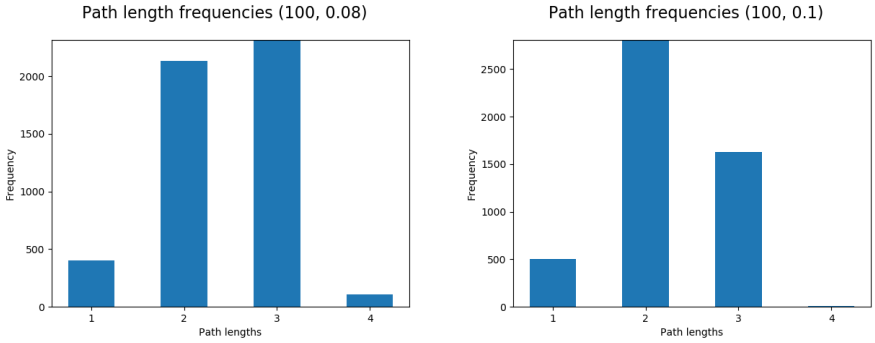

Fig. 3. Histogram of shortest pairwise distances for probability 0.08 and 0.1

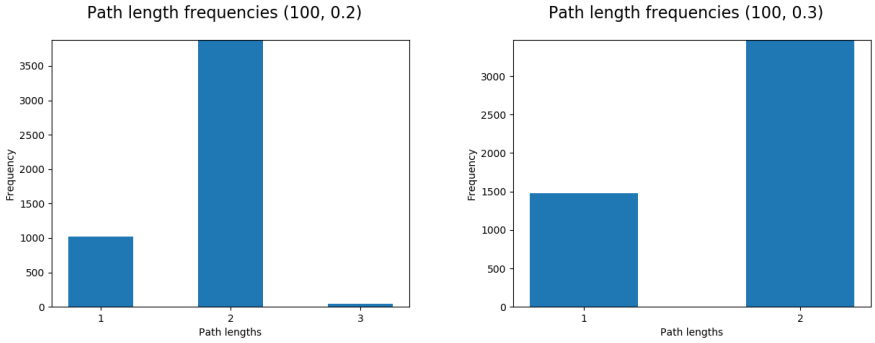

Fig. 4. Histogram of shortest pairwise distances for probability 0.2 and 0.3

3 Mountain car

The mountain car problem is a classic reinforcement learning problem. The agent is a car in a valley and the its goal is to drive to the top of the mountain. The problem is that the gravity acceleration is stronger than the car engine, creating a learning challenge for the agent [1]. The agent's goal is to learn a policy to climb the mountain and reach the goal in the shortest possible time. The state space is described by two continous variables, namely position and velocity. The agent can choose between three actions: move right (towards the goal), move left (opposite to the goal), stay still. The problem can be reduced to a finite Markov decision process where Q-learning algorithm can be used to find the optimal policy to choose an action for a given state. The value of an action (*right, left, no move*) for a given state (*position and velocity*) are kept in a Q-table. This value

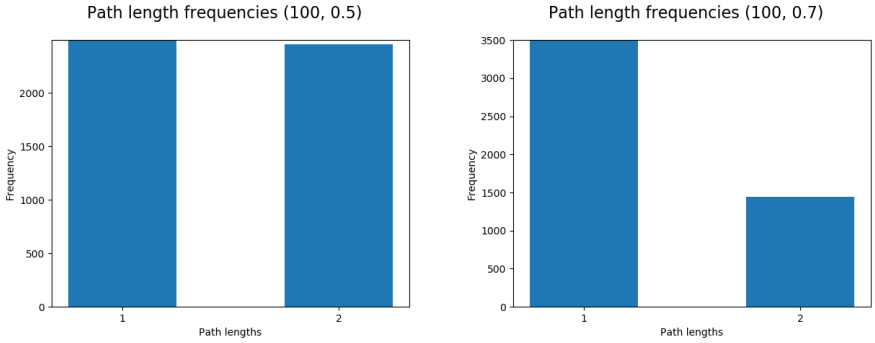

Fig. 5. Histogram of shortest pairwise distances for probability 0.5 and 0.7

gets updated by following the Q-learning value iteration update, see equation 2.

$$Q(S_t, A_t) = (1 - \alpha) \cdot Q(S_t, A_t) + \alpha \cdot (R_t + \gamma \cdot maxQ(S_{t+1}, A)) \tag{2}$$

The Q-learning algorithm works with discrete states. The first step, hence, is to apply state space discretization using tile coding to map continous variables to discrete ones. At each time step the agent's state is translated to a discrete one in order to use Q-learning to update that state's value. Another important aspect of solving the mountain car problem is the exploration probability variable. This variable describes the probability of taking a random action instead of the optimal one. The exploration probability variable importance is given by the fact that allows the agent to search a larger portion of the search state space that can include a better policy. We test the exploration versus exploitation tradeoff by changing the exploration rate and analyze the action value evolution of a specific state (*position and velocity*) through episodes. Figures 6 and 7 show how different exploration probabilities affect the action values. From our

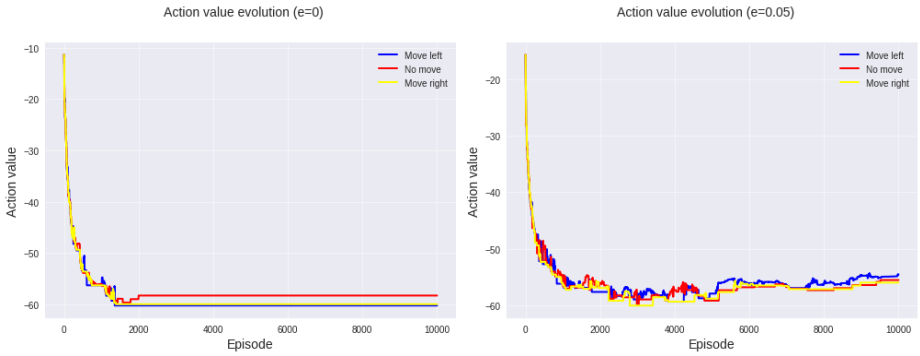

Fig. 6. Action value evolution for exploration probability 0 and 0.05

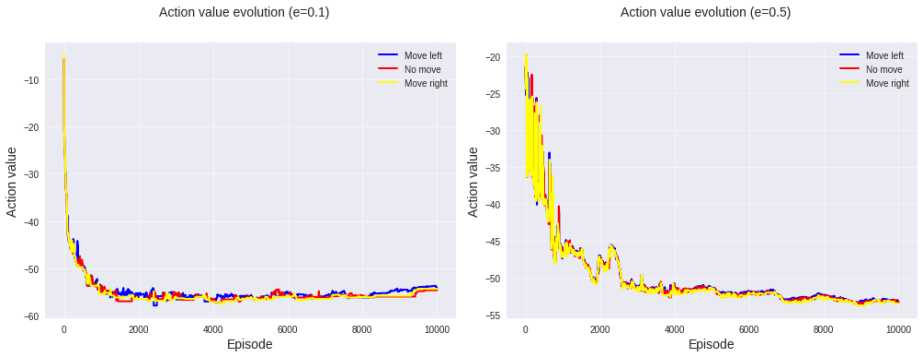

Fig. 7. Action value evolution for exploration probability 0.1 and 0.5

experiments we can deduct that an exploration probability of 0 results in a greedy action selection with an high bias. On the other hand, a really high probability, results in evaluating all the actions for a given state in the same manner. We can, in fact, observe, that with $\epsilon = 0.5$ the action value evolution curves follow the same pattern. This results resemble the underfitting and overfitting errors that occur in supervised machine learning problems. In this case we can conclude that a rate of $\epsilon = 0.05$ or $\epsilon = 0.1$ perform better in the mountain car problem.

References

1. Sutton, R.S. and Barto, A.G., 2011. Reinforcement learning: An introduction.
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2001. Introduction to algorithms second edition. The Knuth-Morris-Pratt Algorithm, year.
3. Weibel, D. (2019). Recursion and Dynamic Programming. [online] Weibeld.net. Available at: http://weibeld.net/algorithms/recursion.html [Accessed 16 Feb. 2019].
4. En.wikipedia.org. (2019). Fibonacci number. [online] Available at: https://en.wikipedia.org/wiki/Fibonacci_number [Accessed 16 Feb. 2019].
5. Sniedovich, M., 2006. Dijkstra's algorithm revisited: the dynamic programming connexion. Control and cybernetics, 35(3), pp.599-620.

4 Appendix A: source code

This appendix section contains the source code of the implemented programs.

4.1 fibonacci.py

```
past_fib = {}


def fibonacci_dynamic(n):
```

```
    if n in past_fib:
        return past_fib[n]
    if n == 1 or n == 2:
        past_fib[n] = 1
        return 1
    total = fibonacci_dynamic(n-1) + fibonacci_dynamic(n-2)
    past_fib[n] = total
    return total


def fibonacci_recursive(n):
    if n == 1 or n == 2:
        return 1
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)


print(fibonacci_dynamic(10))
print(fibonacci_dynamic(30))
print(fibonacci_recursive(10))
print(fibonacci_recursive(30))
```

4.2 graph.py

```
import random
import sys
import numpy as np
import matplotlib.pyplot as plt

graph = {0: {1: 2, 4: 4},
         1: {2: 3},
         2: {3: 5, 4: 1},
         3: {0: 8},
         4: {3: 3}}


def constructGraph(nodes, probability):
  result = np.zeros((nodes, nodes))
  result_dict = {}
  for i in range(nodes):
    partial_dict = {}
    for j in range(nodes):
      if j == i:
        result[i][j] = 0
      elif j < i:
        result[i][j] = result[j][i]
        if result[j][i] > 0:
```

```
          partial_dict[j] = result[j][i]
      else:
        if random.random() <= probability:
          partial_dict[j] = 1
          result[i][j] = 1
        else:
          result[i][j] = 0
      result_dict[i] = partial_dict
  return result, result_dict


def allPairsShortestPath(g):
  nodes = len(g)
  dist = np.zeros((nodes, nodes))
  pred = np.zeros((nodes, nodes))
  for u in g:
    for v in g:
      dist[u][v] = sys.maxsize
      pred[u][v] = None
    dist[u][u] = 0
    pred[u][u] = None
    for v in g[u]:
      dist[u][v] = g[u][v]
      pred[u][v] = u
  for mid in g:
    for u in g:
      for v in g:
        newlen = dist[u][mid] + dist[mid][v]
        if newlen < dist[u][v]:
          dist[u][v] = newlen
          pred[u][v] = pred[mid][v]
  return(dist, pred)


def constructShortestPath(s, t, pred):
  path = [t]
  while t != s:
    try:
      tmp = pred[s][t]
      t = tmp.astype(int)
    except IndexError:
      return None
    if t is None:
      return None
    path.insert(0,t)
```

```
  return path


def constructFrequencyDict(dist):
  result = {}
  for i in range(len(dist)):
    for j in range(len(dist)):
      if i != j and j > i:
       key = dist[i][j]
       if key in result:
         result[key] += 1
       else:
         result[key] = 1
  labels, values = [], []
  for key in sorted(result):
    labels.append(int(key))
    values.append(result[key])
  return labels, values


def plotResults(labels, values, nodes, prob):
  X = np.arange(len(values))
  plt.bar(X, values, width=0.5)
  plt.xticks(X, labels)
  ymax = max(values) + 1
  plt.ylim(0, ymax)
  plt.suptitle("Path␣length␣frequencies␣({},␣{})".format(nodes,
      prob), fontsize=16)
  plt.ylabel('Frequency')
  plt.xlabel('Path␣lengths')
  plt.show()


random.seed(1)
nodes = 100
probabilities = [0.02, 0.05, 0.08, 0.1, 0.2, 0.3, 0.5, 0.7, 1]
for prob in probabilities:
  print(prob)
  graph2, graph2_dict = constructGraph(nodes, prob)
  dist, pred = allPairsShortestPath(graph2_dict)
  labels, values = constructFrequencyDict(dist)
  plotResults(labels, values, nodes, prob)
```