Artificial intelligence - Project 1
- Search problems -

Hitu Octavian, Munteanu Cezar-Lucian

3/11/2021

1 Uninformed search

1.1 Question 1 - Depth-first search

In this section the solution for the following problem will be presented:

"In search.py, implement ***Depth-First search(DFS) algorithm*** *in function depthFirstSearch. Don't forget that DFS graph search is graph-search with the frontier as a LIFO queue(Stack)."*.

1.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```python
def depthFirstSearch(problem):



    startNode = problem.getStartState()
    frontier = util.Stack()
    visitedNodes = []
    frontier.push((startNode, []))

    if problem.isGoalState(startNode):
        return []



    while not frontier.isEmpty():
        currentNode, path = frontier.pop()
        if currentNode not in visitedNodes:
            visitedNodes.append(currentNode)

            if problem.isGoalState(currentNode):
                return path

            for nextNode, action, cost in problem.expand(currentNode):
                newAction = path + [action]
                frontier.push((nextNode, newAction))



    #util.raiseNotDefined()
```

Explanation:

- Algoritmul DFS cauta in adancime solutia noastra. Este un algoritm de tip LIFO deci folosim o structura de tip stiva pentru a putea gasi elementele pe care le vom expanda ulterior. Verificam daca nodul nostru curent nu a fost expandat, iar daca este T il vom scoate din stipa pentru a-l explora. Daca nodul actual nu este visitat il pune in stiva . Vom verifica totodata daca nodul este finalul, iar daca se indelineste conditia vom parcurge drumul de la coada la start.La final vom parcurge nodurile urmatoare , vom crea pentru ele drumul de la tart pana la ele si vom pune noile noduri in stiva.

Commands:

- python3 pacman.py -l tinyMaze -p SearchAgent
  python3 pacman.py -l bigMaze -z .5 -p SearchAgent
  python3 pacman.py -l mediumMaze -p SearchAgent

1.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Is the found solution optimal? Explain your answer.
A1: Algoritmul DFS va expanda nodurile doar dupa modul in care au fost puse ins stiva.Nu va putea alege cea mai scurta cale.Deci nu este optim.

Q2: Run *autograder python autograder.py* and write the points for Question 1.
A2: Question q1: 4/4

1.1.3 Personal observations and notes

1.2 Question 2 - Breadth-first search

In this section the solution for the following problem will be presented:

*"In **search.py**, implement the **Breadth-First search** algorithm in function breadthFirstSearch."*.

1.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "* YOUR CODE HERE *"

    startNode = problem.getStartState()
    frontier = util.Queue()
    visitedNodes = []
    frontier.push((startNode, []))

    if problem.isGoalState(startNode):
        return []

    while not frontier.isEmpty():
        currentNode, path = frontier.pop()
        if currentNode not in visitedNodes:
            visitedNodes.append(currentNode)

```

```python
            if problem.isGoalState(currentNode):
                return path

            for nextNode, action, cost in problem.expand(currentNode):
                newAction = path + [action]
                frontier.push((nextNode, newAction))


    #util.raiseNotDefined()
```

Explanation:

- Algoritmul BFS este identic in mare parte cu cel de DFS. BFS-ul este un algoritm de tip FIFO deci vom folosi o structura de tip coada. Calea va fi diferita de DFS deoarece algortimul ia primul nod pus in coada si il expandeaza.

Commands:

- python3 pacman.py -l tinyMaze -p SearchAgent -a fn=bfs

  python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

  python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=bfs

1.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Is the found solution optimal? Explain your answer.
A1: Algoritmul BFS gaseste mereu cel mai scurt drum de la start la final , deci este optim.

.

Q2: Run autograder *python autograder.py* and write the points for Question 2.
A2: Question q2: 4/4

1.2.3 Personal observations and notes

1.3 Question 3 - Uniform-cost search

In this section the solution for the following problem will be presented:

*"In search.py, implement **Uniform-cost graph search** algorithm in uniformCostSearchfunction"*

1.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```python
def uniformCostSearch(problem):

    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

Explanation:

•

Commands:

•

1.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Compare the results to the ones obtained with DFS. Are the solutions different? Is the number of extended (explored) states smaller? Explain your answer.
A1:

Q2: Consider that some positions are more desirable than others. This can be modeled by a cost function which sets different values for the actions of stepping into positions. Identify in **searchAgents.py** the description of agents StayEastSearchAgent and StayWestSearchAgent and analyze the cost function. Why the cost .5 ** x for stepping into (x,y) is associated to StayWestAgen.
A2:

Q3: Run autograder *python autograder.py* and write the points for Question 3.
A3:

1.3.3 Personal observations and notes

1.4 References

2 Informed search

2.1 Question 4 - A* search algorithm

In this section the solution for the following problem will be presented:

*"Go to aStarSearch in search.py and implement **A* search algorithm**. A* is graphs search with the frontier as a priorityQueue, where the priority is given bythe function g=f+h".*

2.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```python
def aStarSearch(problem, heuristic=nullHeuristic):


    frontier = util.PriorityQueue()

    exploredNodes = []

    startState = problem.getStartState()
    startNode = (startState, [], 0)

    frontier.push(startNode, 0)

    while not frontier.isEmpty():

        currentState, actions, currentCost = frontier.pop()

        currentNode = (currentState, currentCost)
        exploredNodes.append((currentState, currentCost))

        if problem.isGoalState(currentState):
            return actions
        else:
            successors = problem.expand(currentState)
            for sState, sAction, sCost in successors:
                newAction = actions + [sAction]
                newCost = problem.getCostOfActionSequence(newAction)
                newNode = (sState, newAction, newCost)

                already_explored = False

                for explored in exploredNodes:
                    exploredState, exploredCost = explored
                    if( sState == exploredState) and (newCost >= exploredCost):
                        already_explored = True

                if not already_explored:
```

```
                    frontier.push(newNode, newCost + heuristic(sState, problem))
                    exploredNodes.append((sState, newCost))
    return actions

    #util.raiseNotDefined()
```

Explanation:

- Algoritmul A* cauta cel mai scurt drum de la start la final. Acest lucru se face pe baza formulei f=g+h. La fel ca la BFS si aici vom folosi o coada doar ca suma costurilor si cea a valorii euristice a nodului vor influenta alegerea nodurilor care vor fi expandate. Valoarea cea mai mica a functii f va fi cea care va decide.

Commands:

- python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

  python3 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=nullHeuristic

2.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Does A* and UCS find the same solution or they are different?
A1: Nu am facut UCS.

Q2: Does A* finds the solution with fewer expanded nodes than UCS?
A2:

Q3: Does A* finds the solution with fewer expanded nodes than UCS?
A3:

Q4: Run autograder *python autograder.py* and write the points for Question 4 (min 3 points).
A4: Question q3: 4/4

2.1.3 Personal observations and notes

2.2 Question 5 - Find all corners - problem implementation

In this section the solution for the following problem will be presented:

*"Pacman needs to find the shortest path to visit all the corners,regardless there is food dot there or not. Go to **CornersProblem** in searchAgents.py and propose a representation of the state of this search problem. It might help to look at the existing implementation for PositionSearchProblem. The representation should include only the information necessary to reach the goal. Read carefully the comments inside the class CornersProblem."*.

2.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState):

        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"


    def getStartState(self):

        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()

    def isGoalState(self, state):

        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()


    def getSuccessors(self, state):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]

            "*** YOUR CODE HERE ***"

        self._expanded += 1 # DO NOT CHANGE
        return successors


    def getCostOfActions(self, actions):
        if actions == None: return 999999
        x,y= self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
```

```
            if self.walls[x][y]: return 999999
        return len(actions)
```

Explanation:

-

Commands:

-

2.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: For mediumCorners, BFS expands a big number - around 2000 search nodes. It's time to see that A* with an admissible heuristic is able to reduce this number. Please provide your results on this matter. (Number of searched nodes).

A1:

2.2.3 Personal observations and notes

2.3 Question 6 - Find all corners - Heuristic definition

In this section the solution for the following problem will be presented:

"Implement a consistent heuristic for CornersProblem. Go to the function ***cornersHeuristic*** *in searchAgent.py."*.

2.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def cornersHeuristic(state, problem):

    "*** YOUR CODE HERE ***"
    return 0 # Default to trivial solution
```

Explanation:

-

Commands:

-

2.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test with on the mediumMaze layout. What is your number of expanded nodes?
A1:

2.3.3 Personal observations and notes

2.4 Question 7 - Eat all food dots - Heuristic definition

In this section the solution for the following problem will be presented:

"Propose a heuristic for the problem of eating all the food-dots. The problem of eating all food-dots is already implemented in ***FoodSearchProblem*** *in searchAgents.py."*.

2.4.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
def foodHeuristic(state, problem):

    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    return 0
```

Explanation:

-

Commands:

-

2.4.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test with autograder *python autograder.py*. Your score depends on the number of expanded states by A* with your heuristic. What is that number?
A1:

2.4.3 Personal observations and notes

2.5 References

3 Adversarial search

3.1 Question 8 - Improve the ReflexAgent

In this section the solution for the following problem will be presented:

"Improve the ReflexAgent such that it selects a better action. Include in the score food locations and ghost locations. The layout testClassic should be solved more often.".

3.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
class ReflexAgent(Agent):
    """
    A reflex agent chooses an action at each choice point by examining
    its alternatives via a state evaluation function.

    The code below is provided as a guide.  You are welcome to change
    it in any way you see fit, so long as you don't touch our method
    headers.
    """


    def getAction(self, gameState):
        """
        You do not need to change this method, but you're welcome to.

        getAction chooses among the best options according to the evaluation function.

        Just like in the previous project, getAction takes a GameState and returns
        some Directions.X for some X in the set {NORTH, SOUTH, WEST, EAST, STOP}
        """
        # Collect legal moves and child states
        legalMoves = gameState.getLegalActions()

        # Choose one of the best actions
        scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the best

        "Add more of your code here if you want to"

        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState, action):
        """
        Design a better evaluation function here.
```

```

        The evaluation function takes in the current and proposed child
        GameStates (pacman.py) and returns a number, where higher numbers are better.

        The code below extracts some useful information from the state, like the
        remaining food (newFood) and Pacman position after moving (newPos).
        newScaredTimes holds the number of moves that each ghost will remain
        scared because of Pacman having eaten a power pellet.

        Print out these variables to see what you're getting, then combine them
        to create a masterful evaluation function.
        """
        # Useful information you can extract from a GameState (pacman.py)
        childGameState = currentGameState.getPacmanNextState(action)
        newPos = childGameState.getPacmanPosition()
        newFood = childGameState.getFood()
        newGhostStates = childGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

        "* YOUR CODE HERE *"

        newFood = childGameState.getFood().asList()
        minFood = float("inf")
        #extrage cea mai aproape bucatica de mancare
        for food in newFood:
            minFood = min(minFood, manhattanDistance(newPos, food))

        # evita fantoma daca este prea aproape
        for ghost in childGameState.getGhostPositions():
            if (manhattanDistance(newPos, ghost) < 2):
                return -float('inf')

        return childGameState.getScore() + 1.0/minFood

def scoreEvaluationFunction(currentGameState):
    """
    This default evaluation function just returns the score of the state.
    The score is the same one displayed in the Pacman GUI.

    This evaluation function is meant for use with adversarial search agents
    (not reflex agents).
    """
    return currentGameState.getScore()
```

Explanation:

- Pentru inceput , ne-am creat o variabila noua newFood in care am pus bucatelele de mancare ca intr-o lista. Variabila minFood reprezinta minimul de bucatele ramase. Am folosit un for pentru a calcula distanta minima pana la urmatoarea bucatica de mancare cu ajutorul functii manhattandistance la care am folosit ca parametri o posibila viitoare pozitie si datele unei bucatele de mancare.Ulterior am folosit un for pentru a ne asigura ca evitam contactul cu fantoma. In acest for ne-am folosit din nou de functia manhattandistance pentru a calcula distanta de la o posibila viitoare pozitie la fantoma. Ulterior aceasta fiind comparata cu 2 pentru a verifica daca fantoma este prea aproape sau nu.Functia va returna scorul.

Commands:

- python pacman . py -p ReflexAgent

  python pacman . py -p ReflexAgent -l testClassic

3.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test your agent on the openClassic layout. Given a number of 10 consecutive tests, how many types did your agent win? What is your average score (points)?
A1: Am rulat ,iar pacman castiga de fiecare data cu o medie de 1240 de puncte.

3.1.3 Personal observations and notes

3.2 Question 9 - H-Minimax algorithm

In this section the solution for the following problem will be presented:

" Implement H-Minimax algorithm in MinimaxAgentclass from multiAgents.py. Since it can be more than one ghost, for each max layer there are one ormore min layers.".

3.2.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """

    def getAction(self, gameState):
        """
          Returns the minimax action from the current gameState using self.depth
          and self.evaluationFunction.
          Here are some method calls that might be useful when implementing minimax.
          gameState.getLegalActions(agentIndex):
            Returns a list of legal actions for an agent
            agentIndex=0 means Pacman, ghosts are >= 1
          gameState.generateSuccessor(agentIndex, action):
            Returns the successor game state after an agent takes an action
          gameState.getNumAgents():
            Returns the total number of agents in the game
        """
        "* YOUR CODE HERE *"

        # Format of result = [score, action]
```

```
        result = self.minmax_decision(gameState, 0, 0)

        # Return the action from result
        return result[1]

    def minmax_decision(self, gameState, index, depth):
        # aceasta functie va returna perechi de forma [value, action]


        # verificam daca ajunge in stadiu terminal
        if gameState.isWin() or gameState.isLose() or len(gameState.getLegalActions(index)) == 0 or dept
            return gameState.getScore(), ""

        # ramura pentru Pacman
        if index == 0:
            return self.max_value(gameState, index, depth)

        # ramura pentru Ghost
        else:
            return self.min_value(gameState, index, depth)

    def max_value(self, gameState, index, depth):
       # returneaza valoarea maxima pentru multi-agent


        legalMoves = gameState.getLegalActions(index)
        max_value = float("-inf")
        max_action = ""

        for action in legalMoves:
            child = gameState.getNextState(index, action)
            child_index = index + 1
            child_depth = depth

            # daca ajungem la ultima actiune disponibila crestem adancimea si trecem mai departe
            if child_index == gameState.getNumAgents():
                current_value = self.minmax_decision(child, child_index % gameState.getNumAgents(), chil
            else:
                current_value = self.minmax_decision(child, child_index, child_depth)[0]

            if current_value > max_value:
                max_value = current_value
                max_action = action

        return max_value, max_action

    def min_value(self, gameState, index, depth):
        # returneaza valoarea minima pentru multi-agent


        legalMoves = gameState.getLegalActions(index)
        min_value = float("inf")
        min_action = ""

```

```
        for action in legalMoves:
            child = gameState.getNextState(index, action)
            child_index = index + 1
            child_depth = depth

            # daca ajungem la ultima actiune disponibila crestem adancimea si trecem mai departe
            if child_index == gameState.getNumAgents():
                current_value = self.minmax_decision(child, child_index % gameState.getNumAgents(), chil
            else:
                current_value = self.minmax_decision(child, child_index, child_depth)[0]

            if current_value < min_value:
                min_value = current_value
                min_action = action

        return min_value, min_action
```

Explanation:

- Pentru a realiza clasa minimaxAgent am folosit in plus pe langa ce era deja definit 3 functii suplimentare minmaxdecision, maxvalue si minvalue . Pentru inceput functia minmaxdecision selecteeaza pe care dintre functiile de min si max se va merge.Functia maxvalue este formata in principal dintr-un for cu ajutorul caruia vom parcurge toate miscarile legale care ne sunt permise din pozitia curenta . De fiecare data ne aactualizam valoarea maxima si noile actiuni. Tot in acest for verificam daca am parcurs sau nu toata caile posibile astfel construindu-ne pentru fiecare copil o valoare curenta . Acesta functie returneaza valoarea maxima si actiunea de la pozitia respectiva. Analog pentru min max, diferenta stand doar in faptul ca aceasta calculeaza minimul si nu maximul valorii , returnandu-l impreuna cu actiunile de a aceasta pozitie.In afara acestor functii vom apela pentru nodul curent functia minmaxdecision.

Commands:

- python pacman . py -p MinimaxAgent -l minimaxClassic -a depth =4

  python autograder . py -q q2

3.2.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test Pacman on trappedClassic layout and try to explain its behaviour. Why Pacman rushes to the ghost?

A1: In acest caz pacman reactioneaza diferit in functie de adancime. Pentru adancime mai mare sau egala cu 3 pacman nu oberva fantoma albastra si va merge spre cea portocalie. In caz contrar sansele de castig sunt de 50-50 , el descoperind fantoma albastra , mergand dupa ea. Acesta in unele situatii va reusi sa manance toata mancarea, castigand.

3.2.3 Personal observations and notes

3.3 Question 10 - Use $\alpha - \beta$ pruning in AlphaBetaAgent

In this section the solution for the following problem will be presented:

*" Use alpha-beta prunning in **AlphaBetaAgent** from multiagents.py for a more efficient exploration of minimax tree."*.

3.3.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

```
```

Explanation:

-

Commands:

-

3.3.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

Q1: Test your implementation with autograder **python autograder.py** for Question 3. What are your results?
A1:

3.3.3 Personal observations and notes

3.4 References

4 Personal contribution

4.1 Question 11 - Define and solve your own problem.

In this section the solution for the following problem will be presented:

4.1.1 Code implementation

This sub-section is dedicated to showcasing your own solution that you came up with for solving the above question. One has to put here any **code** that has been used for solving the above task, along with **comments** that explain every design decision made. To reference the code, please make use of the *code lines number*. Additionally, complete this sub-section with any **command configurations** that you may have used during the implementation or testing process (please fill in *just the arguments*).

Code:

Explanation:

-

Commands:

-

4.1.2 Questions

This sub-section is dedicated to the additional questions that come along with the exercise. Please answer to the following questions:

4.1.3 Personal observations and notes

4.2 References