Wordle

컴퓨터의 개념 및 실습 (005)

팀	05
팀 원	강성민, 이동준, 이근호

목 차

시행착오

1

Wordle system 구현

1. 입력 단어의 사전 존재 여부 -> 주어진 word.txt를 이용하여 파일로 접근
2. 정답과의 check

    1)같은 자리, 같은 문자 2)사용되지 않은 문자 3)앞에서부터 중복 문자

3. 결과 반환

시행착오

2

Wordle의 특징

1. 대부분의 단어가 모음을 포함하고 있다.

2. 자주 쓰이는 알파벳(e, i, r, s, t, a) 등이 정해져 있다.

3. endgame 때 알파벳 후보들을 모두 포함하는 단어로 test하면 좋다.

4. 가능성이 높은 순이라면 첫 단어가 고정될 것이다.

시행착오

3

Wordle 최적화

1. frequency analysis (암호학 terminology)

: 전체 사전 중에서 각 알파벳이 사용된 횟수를 이용하여 확률적으로

등장하기 쉬운 알파벳부터 우선적으로 검토

: 알파벳별 순위에 따라서 전부 or 가장 많이 포함하고 있는 단어들을 우선적으로 선별

시행착오

frequency(self)

```
alphabet_counter <- {v:[0,0,0,0,0] for v in “abcde…wxyz}

for word in self.possible_set:
    for i, c in enumerate(word):
        alphabet_counter[c][i] += 1

maxScore <- float(‘-inf’)

for word in self.possible_set:
    curr_score <- {}
    for i, c in enumerate(word):
        curr_score[c] <- max(curr_score[c], alphabet_counter[c][i])

    if(maxScore < sum(curr_score.values()):
        maxScore <- sum(curr_score.values())
        maxWord = word

return maxWord
```

시행착오

3

Wordle 최적화

2. distinct한 여러 문자를 사용

: STARE - DOING - LUCKY ..

: 전반적으로 성능이 잘 나오지만, 저격 데이터가 존재

(or distinct한 문자열 조합을 몇 개 찾고서 random하게 선택하는 것도 고려 가능)

시행착오

__init__(self)

```
self.wordList = ["lucky", "doing"]
self.word = 'stare'
```

guess(self, prev)

```
self.converge_possible_set(self.word, prev)
if self.wordList:
        self.word = self.wordList.pop()
        return self.word
```

시행착오

3

Wordle 최적화

3. 남아있는 후보들에 대해서 그룹에 대한 scoring

: 좋은 partition일수록, log scale로 계산할 수 있을 듯

: 초기 단어는 따로 제공하여, possible set을 크기를 많이 제거하고 시작

시행착오

_partition(self, comp_word)

```
possibility = [0 for _ in range(243)] #(=3^5 for distinct pattern)
    for possible_word in self.possible_set:
    int_pattern = revert_pattern(compare(possible_word, comp_word))
    possibility[int_pattern] += 1
return sorted(possibility)
```

_average(self)

_minMax(self)

_variance(self)

시행착오

3

Wordle 최적화

4. 모음 기반 find - - audio, adieu

: 모음부터 조사하고 가능한 자음을 나중에 해결

: 자음만 존재하는 단어 / 비슷한 꼴의 단어

ex) ?LA?E // 가능한 자음들 조합만 너무 많음

시행착오

3

Wordle 최적화

5. j-th character (1 < = j < = 5) 로 가능한 알파벳을 관리

: 매 query마다 각 자리에 들어갈 수 있는 알파벳을 만족하면서 단어를 select

시행착오

_position_manage(self)

```
guess = ['' for _ in range(_wordLength)]
while ''.join(guess) not in wordSet:
    for i in range(_wordLength):
        guess[i] = random.choice(tuple(self.positions[i]))
return ''.join(guess)
```

__converge_position(self, word, result)

```
for i in range(_wordLength):
    if result[i] == "B":
        self.positions[i] = set(word[i])
    elif result[i] == "Y":
        self.positions[i].discard(word[i])
    else:
        for j in range(_wordLength):
            self.positions[j].discard(word[i])
```

시행착오

3

Wordle 최적화

6. score 위주로 코딩?

1) Grey는 포함하지 않음

2) B, Y를 포함하면 점수 부여

3) 새로운 정보면 더 높은 점수

시행착오

3

Wordle 최적화

> time complexity를 줄이는 idea?

: 확정된 위치를 제외하고, 나머지 부분에 대해서 빈도 계산

> letter frequency로 query가 많이 필요한 단어

: 남은 한 자리 자음을 찾는 경우가 많음

시행착오

4

프로젝트 진행 계획

: 시간 초과가 걸리지 않는다는 전제 하에, 3-3) idea_partitioning으로 진행

a. 가장 많이 등장하는 패턴의 확률의 min.

b. 패턴의 확률에 대한 average

c. 패턴의 확률 variance를 기준으로 evaluate

--- > 시간초과 문제로 기각

시행착오

4

프로젝트 진행 계획

: 다양한 방법의 scoring으로 선회

1) distinct, green, yellow, grey에 따라 차등 점수 부여

2) 가능한 단어목록에 대하여 각 자리별 해당 알파벳의 등장횟수 합으로 점수 부여

2)-1 한번이라도 Green을 받은 적이 있는 자리는 score에 계산 X

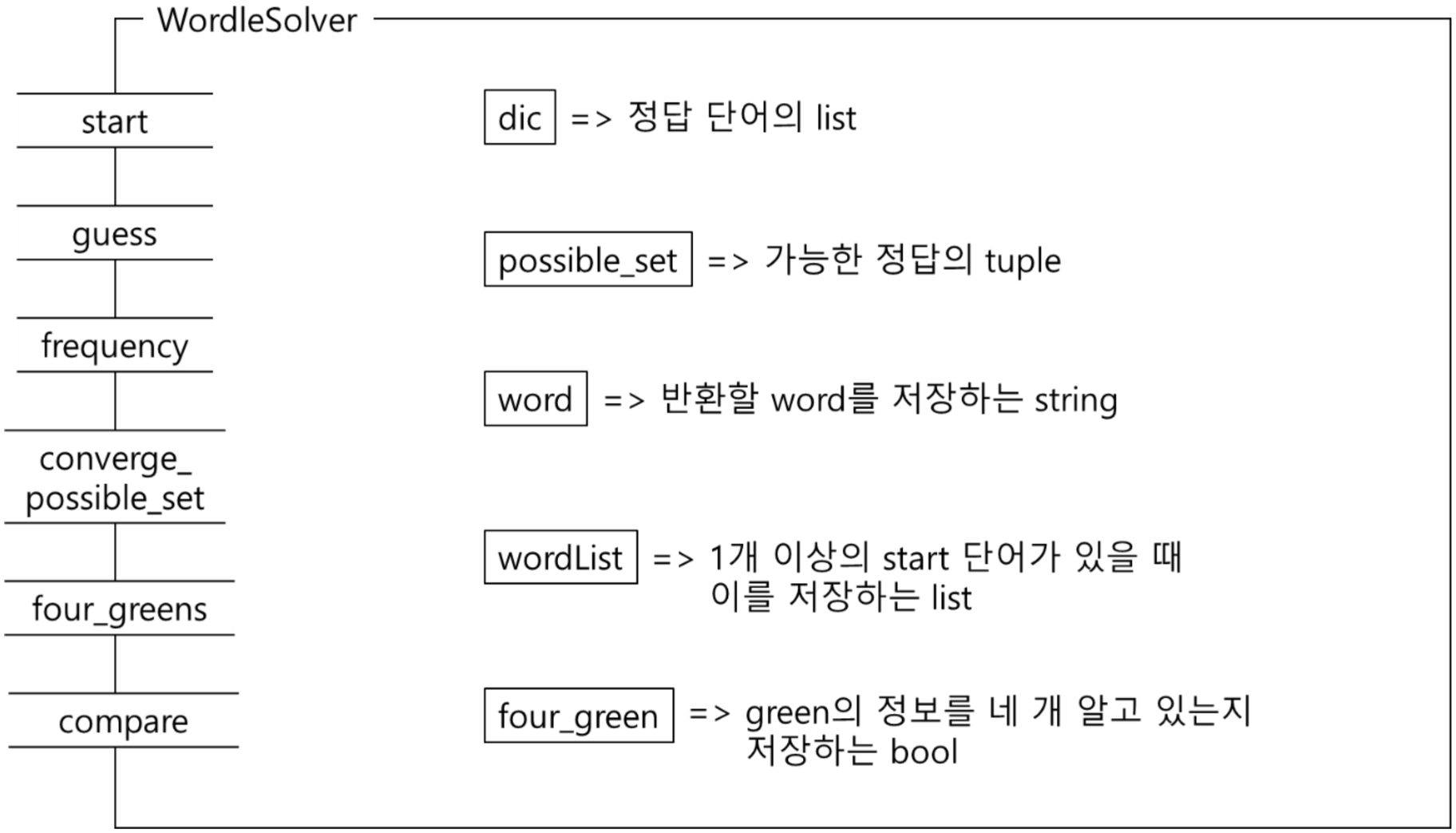
시행착오

4

프로젝트 진행 계획

: 자리별로 가능한 알파벳의 목록을 따로 관리

코드 설명


WordleSolver
start
guess
frequency
converge_
possible_set
four_greens
compare
dic => 정답 단어의 list
possible_set => 가능한 정답의 tuple
word => 반환할 word를 저장하는 string
wordList => 1개 이상의 start 단어가 있을 때
이를 저장하는 list
four_green => green의 정보를 네 개 알고 있는지
저장하는 bool

코드 설명

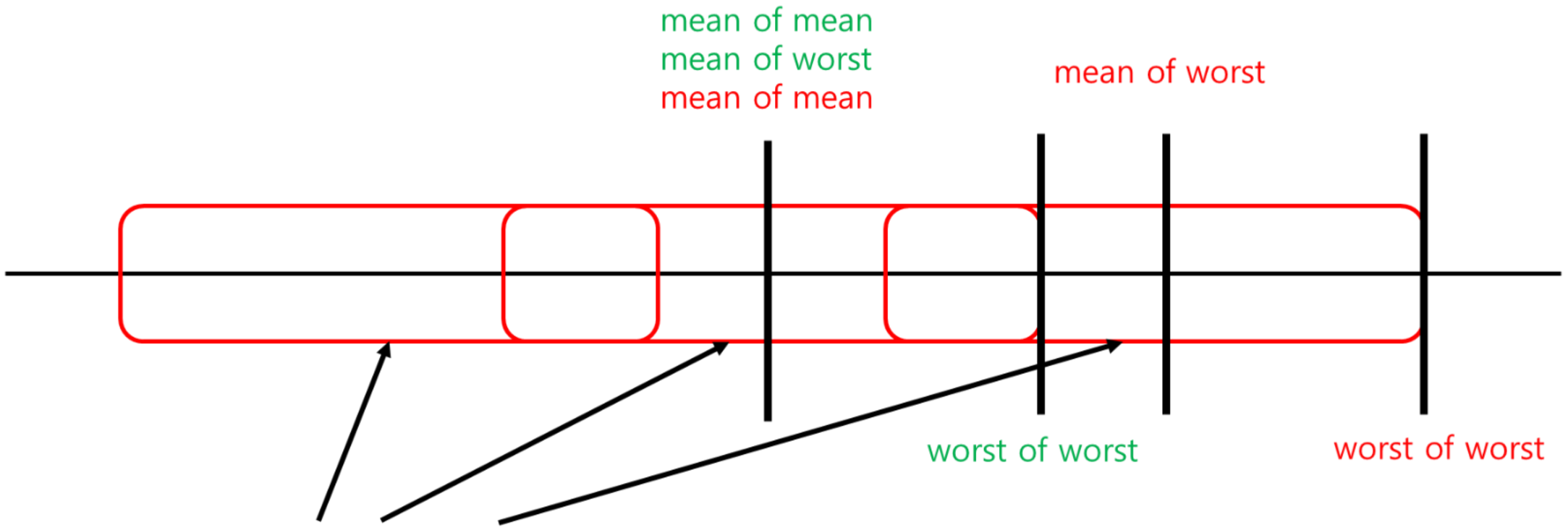
0

랜덤성 제거

과제에서 평가하는 것은 mean of mean, mean of worst, worst of worst

랜덤성이 있으면 mean of worst, worst of worst가 나빠지는 결과

코드 설명



- 랜덤성 없을 때 기댓값
- 랜덤성 있을 때 기댓값

코드 설명

1

start 작동 방식

stare -〉 doing -〉 lucky의 start 단어

average query는 약간 증가하지만,

worst query는 크게 감소함.

코드 설명

2

guess 작동 방식

1. self.possible_set 업데이트

2. self.wordList에 원소가 있으면 마지막 원소를 삭제하고 반환

3. four_green이 True면 four_greens()의 결과 반환

4. frequency의 결과로 나온 word를 반환

코드 설명

start(self)

```
return self.word
```

guess(self, prev)

```
self.converge_possible_set(self.word, prev)

if prev.count('B') = 4 and len(self.possible_set) > 3:
    self.four_green = True

if wordList is not empty:
    self.word <- wordlist.pop
    return self.word

if self.four_green = True:
    self.word <- self.four_greens()
else:
    self.word <- self.frequency()

return self.word
```

코드 설명

3

frequency 작동 방식

1. possible_set의 각 word에 대해 문자의 위치 별 frequency 계산

2. possible_set의 각 word에 대해 각 문자의 frequency 총합을 계산함. 이때, 중복된 문자는 계산하지 않음

(ex. steep에서 e는 위치 2, 3의 frequency 중 max값으로 한 번만 계산됨)

3. frequency 총합이 가장 큰 word를 return함

코드 설명

frequency(self)

```
alphabet_counter <- {v:[0,0,0,0,0] for v in “abcde…wxyz}

for word in self.possible_set:
    for i, c in enumerate(word):
        alphabet_counter[c][i] += 1

maxScore <- float(‘-inf’)

for word in self.possible_set:
    curr_score <- {}
    for i, c in enumerate(word):
        curr_score[c] <- max(curr_score[c], alphabet_counter[c][i])

    if(maxScore < sum(curr_score.values()):
        maxScore <- sum(curr_score.values())
        maxWord = word

return maxWord
```

코드 설명

4

converge_possible_set, compare 작동 방식

candidate list 선언

self.possible_set의 word들 중 정답으로 가능한 word를 업데이트

compare는 이 작업을 도와주는 staticmethod

코드 설명

converge_possible_set(self, word, result)

```
candidate <- []

for possible_answer in self.possible_set:
    if compare(word, possible_answer) = result:
        candidate.append(possible_answer)

self.possible_set <- tuple(candidate)
```

compare(comp, crit)

```
crit을 정답이라고 가정하고 comp를 Wordle 방식으로 evaluate


return result
```

코드 설명

5

four_greens 작동 방식

green이 아닌 index에 들어갈 수 있는 문자 계산 후 priority_letters에 저장

전체 answer set에서 priority_letters의 문자가 많이 포함된 정도로 score 계산

가장 큰 score를 가지는 word return

코드 설명

four_greens(self)

```
    self.four_green <- False

    prev <- self.possible_set[0]
    word <- self.possible_set[1]
    for i in range(5):
        if prev[i] != word[i]:
            idx <- i

    priority_letters <- set()
    for word in self.possible_set:
        priority_letters.add(word[idx])



    max_score <- 0
    max_word <- ''
```

```
for word in self.dic:
    curr_score <- 0
    used_letter <- set()

    for letter in word:
        if letter in used_letter:
            continue
        used_letter.add(letter)
        if letter in priority_letters:
            curr_score += 1

    if max_score < curr_score:
        max_score <- curr_score
        max_word <- word

return max_word
```

실행 결과

1

요약

평균 실행 횟수 : 4.79회

최소 실행 횟수 : 1회

최다 실행 횟수 : 10회