15: Backtracking
backtracking
Backtracking

• A general algorithm for finding all (or some) solutions to a computational problem that
  – incrementally builds candidates to the solution
  – selects a candidate to check
  – abandons the candidate when it finds cannot lead to a solution, then backtracks to prior candidates

• A backtracking algorithm either finds the solution or exhaustively searches all possibilities before failing to find a solution
Backtracking

• There are two principal techniques for implementing backtracking algorithms
  – one uses recursion
  – one uses stacks
backtracking with recursion
Backtracking using recursion

• Like most recursive algorithms, the execution of a backtracking algorithm can be illustrated using a tree
  – the root of the tree is the first call to the algorithm
  – the edges in the tree are the recursive calls
  – the nodes at a given level are the candidates up to that point
  – the leaves are either solutions or dead ends
Backtracking recursion tree

![Recursion Tree Diagram]

A

B

D bad

E bad

C

F good!

G bad
Backtracking recursion tree

Visiting A is the first call to the algo
- there are two candidates: B and C
The call to B is a recursive call to check that solution
- if B is not the solution, there are two candidates: D and E
The call to D is a recursive call to check that solution
D is not a solution: *backtrack* to B

*and so on…*
Backtracking recursion tree

Note:
• there is not an actual tree data structure
• the tree is an abstract model of the possible sequences of choices the algorithm makes
Backtracking algorithm: example
Word morph

• Change one word into another by changing one letter at a time

Examples:

- cat/dog
  - cat → cot → cog → dog

- head/tail
  - head → heal → hell → hall → tall → tail

• Also called Word Ladder
Word morph

• Change one word into another by changing one letter at a time

Examples:

  cat/dog
  – cat → cot → cog → dog

  head/tail
  – head → heal → hell → hall → tall → tail

• Imagine a tree where each level is the set of possible words created by changing one character
Word morph: sample tree

one letter different from prior level
Word morph: sample tree

- How do we generate and search the possibilities?
Word morph

• Change one word into another by changing one letter at a time

• All of the words generated by changing one letter go in the next level of the tree: head

hea_  he_d  h_ad  _ead
  ↓    ↓    ↓    ↓
 heal  held  hoad  mead
 heap  bead  lead
 heat  ...
 ...
Word morph

• Given a dictionary of valid words
  – Generate the set of words that differ from a word $w1$ by one letter

• Solution 1
  – For each position $i$ in $w1$,
    for each letter in the alphabet,
    create a new word by changing position $i$ to the next letter in the alphabet
    if it's in the dictionary, add it to the set of words*

*unless it's been seen already
Word morph

• Given a dictionary of valid words
  ‒ Generate the set of words that differ from a word \( w_1 \) by one letter

• Solution 2
  ‒ Write a distance function that computes the number of positions in two strings where the two strings differ
    - distance(heap, heat) returns 1
    - distance(keep, beet) returns 2

  ‒ For each word \( w_2 \) in the dictionary
    if the distance between \( w_1 \) and \( w_2 \) is 1, then add \( w_2 \) to the set of words
      * unless it's been seen already
Exercise

Write a function `dist(w1, w2)` that returns the number of positions where words `w1` and `w2` differ. It requires that `len(w1) == len(w2).

Use an assert to verify the lengths of `w1` and `w2` are the same.

Use a list comprehension in your function.
def main():
    (word1, word2, word_list) = read_input()
    morph_seq = morph(word1, word2, word_list, [])
    print_seq(morph_seq)
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1, w2, word_list):
        for candidate in candidate_list:
            next = candidate
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []:
                return [w1] + result
        return []
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1,w2,word_list):
        for candidate in candidate_list:
            next = candidate
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []:
                return [w1] + result
        return []

Word morph: search

update the current “candidate”
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1,w2,word_list):
        for candidate in candidate_list:
            next = candidate
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []:
                return [w1] + result
        return []

Word morph: search

candidate_list = next_words(w1,w2,word_list):
    for candidate in candidate_list:
        next = candidate
        result = morph(next, w2, word_list, Seen + [w1])
        if result != []:
            return [w1] + result
        return []

search from candidate
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1, w2, word_list):
        for candidate in candidate_list:
            next = candidate
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []:
                return [w1] + result
        return []
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1, w2, word_list):
        for candidate in candidate_list:
            next = candidate
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []:
                return [w1] + result
        return []

Word morph: search

No solution found from the list of candidates; “Backtrack” to prior level
def next_words(wd1, wd2, word_list):
    cands = [wd for wd in word_list \n             if len(wd) == len(wd1) and dist(wd, wd1) == 1]
    cands.sort(key = lambda wd : dist(wd, wd2))

# dist(w1, w2) returns the number of positions where words
# w1 and w2 differ. It requires that len(w1) == len(w2).

def dist(w1, w2):
    assert len(w1) == len(w2)
    diffs = [i for i in range(len(w1)) if w1[i] != w2[i]]
    return len(diffs)
# File: “morph.py”
# Author: Saumya Debray
import sys
from copy import *
DICT = 'WORDS.txt'
def read_input():
    # read the dictionary into a list
    try:
        dict_file = open(DICT)
    except IOError:
        print('ERROR: could not open file: ' + dict_filename)
        sys.exit(1)

    word_list = []
    for word in dict_file:
        word_list.append(word.strip())

    # read the two words to be morphed
    word1 = input('Word 1: ')  
    word2 = input('Word 2: ')  
    return (word1, word2, word_list)

def dist(w1, w2):
    assert len(w1) == len(w2)
    diffs = [i for i in range(len(w1)) if w1[i] != w2[i]]
    return len(diffs)

def dist(w1, w2):
    assert len(w1) == len(w2)
    diffs = [i for i in range(len(w1)) if w1[i] != w2[i]]
    return len(diffs)

def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidates = [w for w in word_list \ 
                       if len(w) == len(w1) and dist(w, w1) == 1]

        # consider candidates closer to w2 first
        candidates.sort(key = lambda w: dist(w, w2))
        for cand in candidates:
            result = morph(cand, w2, word_list, Seen + [w1])

            # a non-empty result means a successful morph
            if result != []:  
                return [w1] + result
        return []

def print_seq(word_list):
    if word_list == []:  
        print('Sorry, no morph sequence found')
    else:
        out_str = ' --> '.join(word_list)
        print(out_str)

def main():
    (word1, word2, word_list) = read_input()
    morph_seq = morph(word1, word2, word_list, [])
    print_seq(morph_seq)
    main()
lambda expressions

• Anonymous functions can be made with the lambda keyword
  – lambda a, b : a + b

• This is a syntactic short hand

• Restricted to single expressions

• Lambda functions can be used wherever a function object is valid
lambda expressions

• Example:

```python
>>> def make_incrementer(n):
    return lambda x: x + n

>>> f = make_incrementer(10)
>>> f
<function make_incrementer.<locals>.<lambda> at 0x10570bf28>
>>> f(2)
12
>>> f(100)
110
```
Word morph: example runs

• cat → dog
  - cat, cot, cog, dog

• head → tail
  - head, heal, heel, hell, hall, tall, tail

• nose → chin
  - nose, Bose, dose, dole, dale, dame, came, cage, cake, cape, care, card, carp, camp, lamp, lame, fame, fare, dare, darn, damn, dawn, down, gown, sown, soon, coon, coin, chin

why the extra words? 😞
Word morph: example runs

depth-first search!
Word morph: example runs

breadth-first search
Challenge

• This version of the word morph game works with just one single word

• What would it take to let the program work with more than one word?
  – keep total length the same

  e.g.: software → soft are → soft ear

• Looking for:

  Wildcats → Beat ASU
Game tree search revisited

Recall our tic-tac-toe program

Given a starting position,
- it generates successive positions from different possible moves
- evaluates the effect of continuing play from each of these positions
- options:
  - pick a move that leads to the best position after some number of turns \( n \) (\( n = \) “lookahead”)
  - search exhaustively for a solution
Game tree search revisited

Recall our tic-tac-toe program

Given a starting position,

- it generates successive positions from different possible moves
- evaluates the effect of continuing play from each of these positions
- options
  - pick a move that leads to the best position after some number of turns $n$ ($n =$ "lookahead")
  - search exhaustively for a solution $\leftarrow$
Game tree search revisited

How do we generate the next possible position for x?

– iterate through the grid and find the next empty spot
– modify the board
Game tree search revisited

How many new board positions are there at this point for x?

- 7 board positions
- search possibilities from left-most at board 1
- if no win is found, start again at board 2
  • "backtrack" to 2
eval_pos(pos, turn)

if game is over with this position board pos
    return "win" or "loss" indication
else
    while there are still open positions
        new_pos = move the player to the next generated position
        result = eval_pos(new_pos, change player)
        if result is win
            return "win" indication
def eval_pos(pos, turn):
    if game_over(pos):
        return win_or_loss(pos, turn)
    else:
        while next_pos != None:
            next_pos = generate_next_pos(pos, turn)
            if next_pos != None:
                result = eval_pos(next_pos, next[turn])
    ...

def generate_next_pos(pos, turn):
    for i in range(3):
        for j in range(3):
            if pos[i][j] == ' ':
                pos[i][j] = turn
                return pos
    return None

Game tree search revisited

\[
\begin{array}{ccc}
    X & O & X \\
    0 & 0 & x \\
\end{array}
\]

next = \{'X': 'O', 'O': 'X'\}
def eval_pos(pos, turn):
    if game_over(pos):
        return win_or_loss(pos, turn)
    else:
        while next_pos != None:
            next_pos = generate_next_pos(pos, turn)
            if next_pos != None:
                result = eval_pos(next_pos, next[turn])

def generate_next_pos(pos, turn):
    return pos

Game tree search revisited
Game tree search revisited

Because arguments are passed by object reference:

- changes made to the board here will be visible here

- but these board positions are supposed to be independent!
Game tree search revisited

Solution: create a copy of the board position

A refresher on copying:

```python
>>> x = [[1,2,3],[4,5,6]]
>>> y = x
>>> y[0].append(73)
>>> x
[[1, 2, 3, 73], [4, 5, 6]]

>>> from copy import *
>>> x = [[1,2,3],[4,5,6]]
>>> y = deepcopy(x)
>>> y[0].append(73)
>>> y
[[1, 2, 3, 73], [4, 5, 6]]
>>> x
[[1, 2, 3], [4, 5, 6]]
```
Game tree search revisited

Solution: create a copy of the board position

```python
from copy import *
...

def generate_next_pos(pos, turn):
    new_pos = deepcopy(pos)
    for i in range(3):
        for j in range(3):
            if new_pos[i][j] == ' ':
                new_pos[i][j] = turn
    return new_pos

return None
```

Updates to `new_pos` don't change `pos`
Recursive search problems: example
The Cracker Barrel peg game

Repeatedly jump over and remove pegs until there is just one peg left on the board
The Cracker Barrel peg game
The Cracker Barrel peg game

# the recursive search

def solve(board, npins, movelist):
    if npins == 1:  # success!
        print_moves(movelist)
    else:
        mvs = [mv for mv in moves if legal_move(board, mv)]
        for mv in mvs:
            newboard = update_board(board, mv)
            solve(newboard, npins-1, movelist+[mv])
The Cracker Barrel peg game

# the recursive search

def solve(board, npins, movelist):
    if npins == 1:  # success!
        print_moves(movelist)
    else:
        mvs = [mv for mv in moves if legal_move(board, mv)]
        for mv in mvs:
            newboard = update_board(board, mv)
            solve(newboard, npins-1, movelist+[mv])

search from the new position
The Cracker Barrel peg game

# moves is a list of the possible moves. An entry (src,mid,dst) # indicates a move from src to dst jumping over mid.

moves = [(0,1,3),(0,2,5),(1,3,6),(1,4,8),
         (2,4,7),(2,5,9),(3,1,0),(3,4,5),(3,6,10),
         (3,7,12),(4,7,11),(4,8,13),(5,2,0),
         (5,4,3), (5,8,12),(5,9,14),(6,3,1),
         (6,7,8),(7,4,2),(7,8,9),(8,4,1),(8,7,6),
         (9,5,2),(9,8,7),(10,6,3),(10,11,12),
         (11,7,4),(11,12,13),(12,7,3),(12,8,5),
         (12,11,10),(12,13,14),(13,8,4),
         (13,12,11),(14,9,5),(14,13,12)]
The Cracker Barrel peg game

# The board is a dictionary \{k_1:v_1, ..., k_n:v_n\} where the \(k_i\) are 
# positions and the \(v_i\) are 0 or 1 indicating whether the 
# position is occupied.

def update_board(board, move):
    (src,mid,dst) = move
    newboard = copy(board)
    newboard[src] = 0  # pin moved away from this position
    newboard[mid] = 0  # pin removed from this position
    newboard[dst] = 1  # pin lands on this position
    return newboard
The Cracker Barrel peg game

def main():
    if len(sys.argv) < 2:
        print("Usage: crackerbarrel.py config_string")
        sys.exit(1)

    # "011111111111111" --> 29,760 solutions
    initial_config = sys.argv[1]
    board = build_board(initial_config)
    npins = initial_config.count('1')
    solve(board, npins, [])

50
The Cracker Barrel peg game:

code

```python
# File: crackerbarrel.py
# Author: Saumya Debray

import sys
from copy import *

""" This program computes all solutions the "Cracker-Barrel Problem, ignoring symmetries. """

# moves is a list of the possible moves. An entry (src,mid,dst) indicates
# a move from src to dst jumping over mid.

moves = [(0,1,3),(0,2,5),(1,3,6),(1,4,8),(2,4,7),(2,5,9),
(3,1,0),(3,4,5),(3,6,10),(3,7,12),(4,7,11),(4,8,13),
(5,2,0),(5,4,3),(5,8,12),(5,9,14),(6,3,1),(6,7,8),
(7,4,2),(7,8,9),(8,4,1),(8,7,6),(9,5,2),(9,8,7),
(10,6,3),(10,11,12),(11,7,4),(11,12,13),
(12,7,3),(12,8,5),(12,11,10),(12,13,14),
(13,8,4),(13,12,11),(14,9,5),(14,13,12)]

# build_board(config) returns a board corresponding to the
# configuration string config. The board is a dictionary
# {k1:v1, ..., kn:vn} where the ki are positions and the vi are 0 or 1  # indicating whether that position is occupied.

def build_board(config):
    n = len(config)
    occupancy_list = [int(k) for k in list(config)]
    return dict(zip(range(n), occupancy_list))

# update_board(board, (src,mid,dst)) returns a new board that gives
# the result of making a move (src,mid,dst), i.e., moving a pin from
# position src to position dst and thereby removing the pin at position
# mid, in the given board.

def update_board(board, move):
    (src,mid,dst) = move
    newboard = copy(board)
    newboard[src] = 0
    newboard[mid] = 0
    newboard[dst] = 1
    return newboard

# legal_move(board, (src,mid,dst)) returns True if (src,mid,dst) is a
# legal move in the given board; False otherwise.

def legal_move(board, mov):
    (src,mid,dst) = mov
    return (board[src] == 1 and board[mid] == 1 and board[dst] == 0)

# solve() performs a brute-force exploration of the search space.

def solve(board, npins, movelist):
    if npins == 1:
        # success!
        print_moves(movelist)
    else:
        mvs = [mv for mv in moves if legal_move(board, mv)]
        for mv in mvs:
            newboard = update_board(board, mv)
            solve(newboard, npins-1, movelist+[mv])

def print_moves(L):
    line = ""
    for i in range(len(L)):
        (src,mid,dst) = L[i]
        line = line + "[" + str(src) + "->" + str(dst) + "]"
    print(line)

# legal move (src,mid,dst) returns True if (src,mid,dst) is a legal move in the given board; False otherwise.

def legal_move(board, mov):
    (src,mid,dst) = mov
    return (board[src] == 1 and board[mid] == 1 and board[dst] == 0)

# solve() performs a brute-force exploration of the search space.

def solve(board, npins, movelist):
    if npins == 1:
        # success!
        print_moves(movelist)
    else:
        mvs = [mv for mv in moves if legal_move(board, mv)]
        for mv in mvs:
            newboard = update_board(board, mv)
            solve(newboard, npins-1, movelist+[mv])

def print_moves(L):
    line = ""
    for i in range(len(L)):
        (src,mid,dst) = L[i]
        line = line + "[" + str(src) + "->" + str(dst) + "]"
    print(line)

# main():
if len(sys.argv) < 2:
    print("Usage: crackerbarrel.py config_string")
sys.exit(1)

# "01111111111111" --> 29,760 solutions
initial_config = sys.argv[1]
board = build_board(initial_config)
npins = initial_config.count('1')
solve(board, npins, [])

main()
```

# legal_move(board, (src,mid,dst)) returns True if (src,mid,dst) is a
# legal move in the given board; False otherwise.

def legal_move(board, mov):
    (src,mid,dst) = mov
    return (board[src] == 1 and board[mid] == 1 and board[dst] == 0)

# solve() performs a brute-force exploration of the search space.

def solve(board, npins, movelist):
    if npins == 1:
        # success!
        print_moves(movelist)
    else:
        mvs = [mv for mv in moves if legal_move(board, mv)]
        for mv in mvs:
            newboard = update_board(board, mv)
            solve(newboard, npins-1, movelist+[mv])

def print_moves(L):
    line = ""
    for i in range(len(L)):
        (src,mid,dst) = L[i]
        line = line + "[" + str(src) + "->" + str(dst) + "]"
    print(line)

# main():
if len(sys.argv) < 2:
    print("Usage: crackerbarrel.py config_string")
sys.exit(1)

# "01111111111111" --> 29,760 solutions
initial_config = sys.argv[1]
board = build_board(initial_config)
npins = initial_config.count('1')
solve(board, npins, [])

main()
The Cracker Barrel peg game

- This formulation of the set of possible moves is specific to a game with 5 rows of pins
- What would a generalization to n rows look like?

# moves is a list of the possible moves. An entry (src,mid,dst) # indicates a move from src to dst jumping over mid.

moves = [(0,1,3),(0,2,5),(1,3,6),(1,4,8),
(2,4,7),(2,5,9),(3,1,0),(3,4,5),(3,6,10),
(3,7,12),(4,7,11),(4,8,13),(5,2,0),
(5,4,3),(5,8,12),(5,9,14),(6,3,1),
(6,7,8),(7,4,2),(7,8,9),(8,4,1),(8,7,6),
(9,5,2),(9,8,7),(10,6,3),(10,11,12),
(11,7,4),(11,12,13),(12,7,3),(12,8,5),
(12,11,10),(12,13,14),(13,8,4),
(13,12,11),(14,9,5),(14,13,12)]