CSc 120

Introduction to Computer Programming II

16: 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

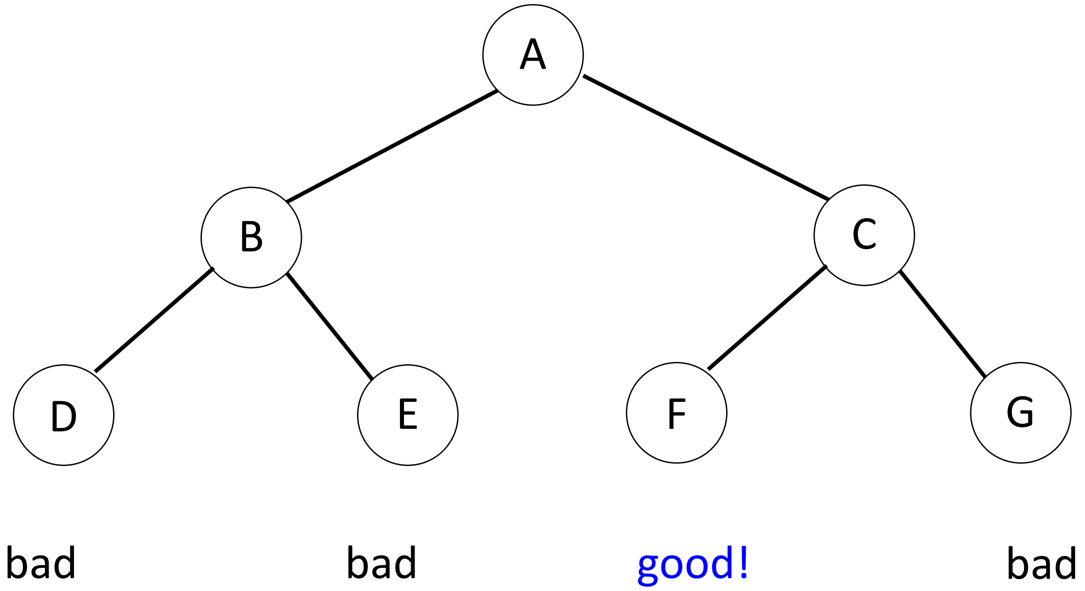
- There are two principal techniques for implementing backtracking algorithms
  - one uses recursion
  - one uses stacks (an explicit stack)

backtracking with recursion

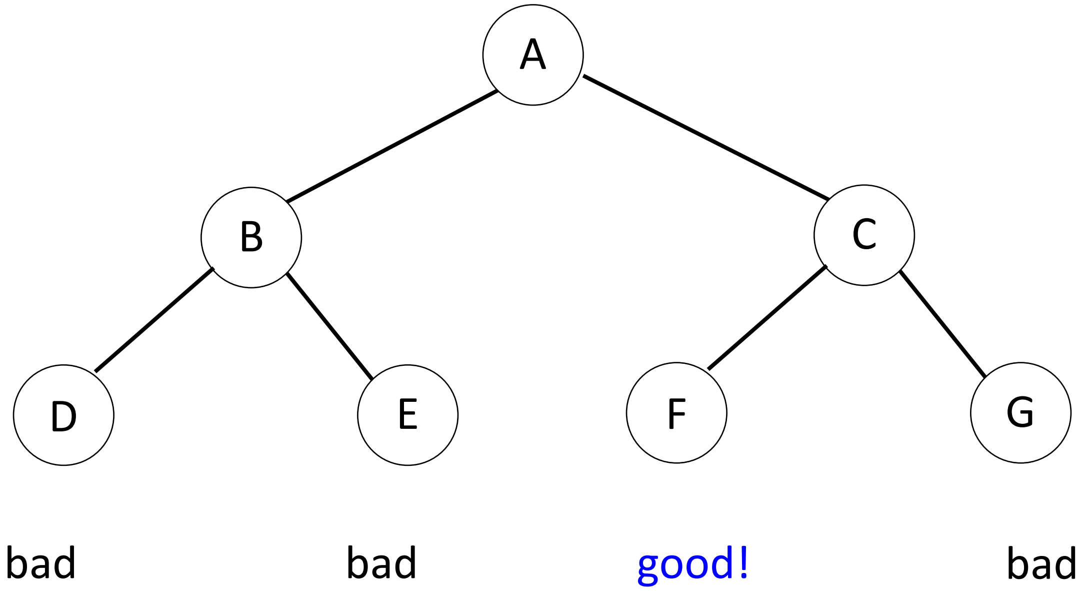
Backtracking using recursion

- 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



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

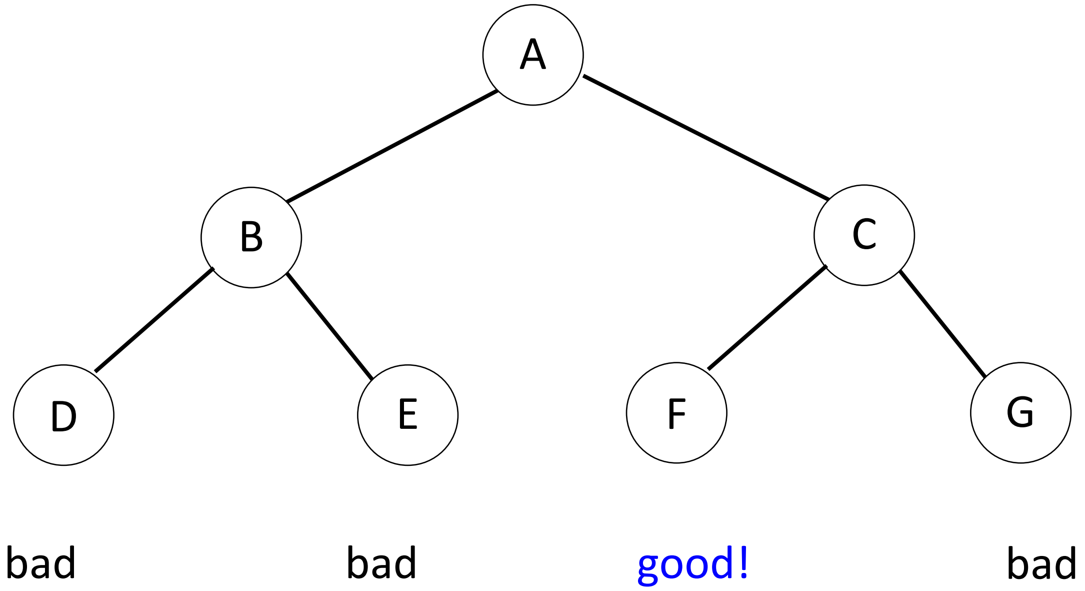
- if B is not the solution, there are two candidates: D and E

The call to D is a recursive call to check that solution

D is not a solution; there are no other candidates; *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

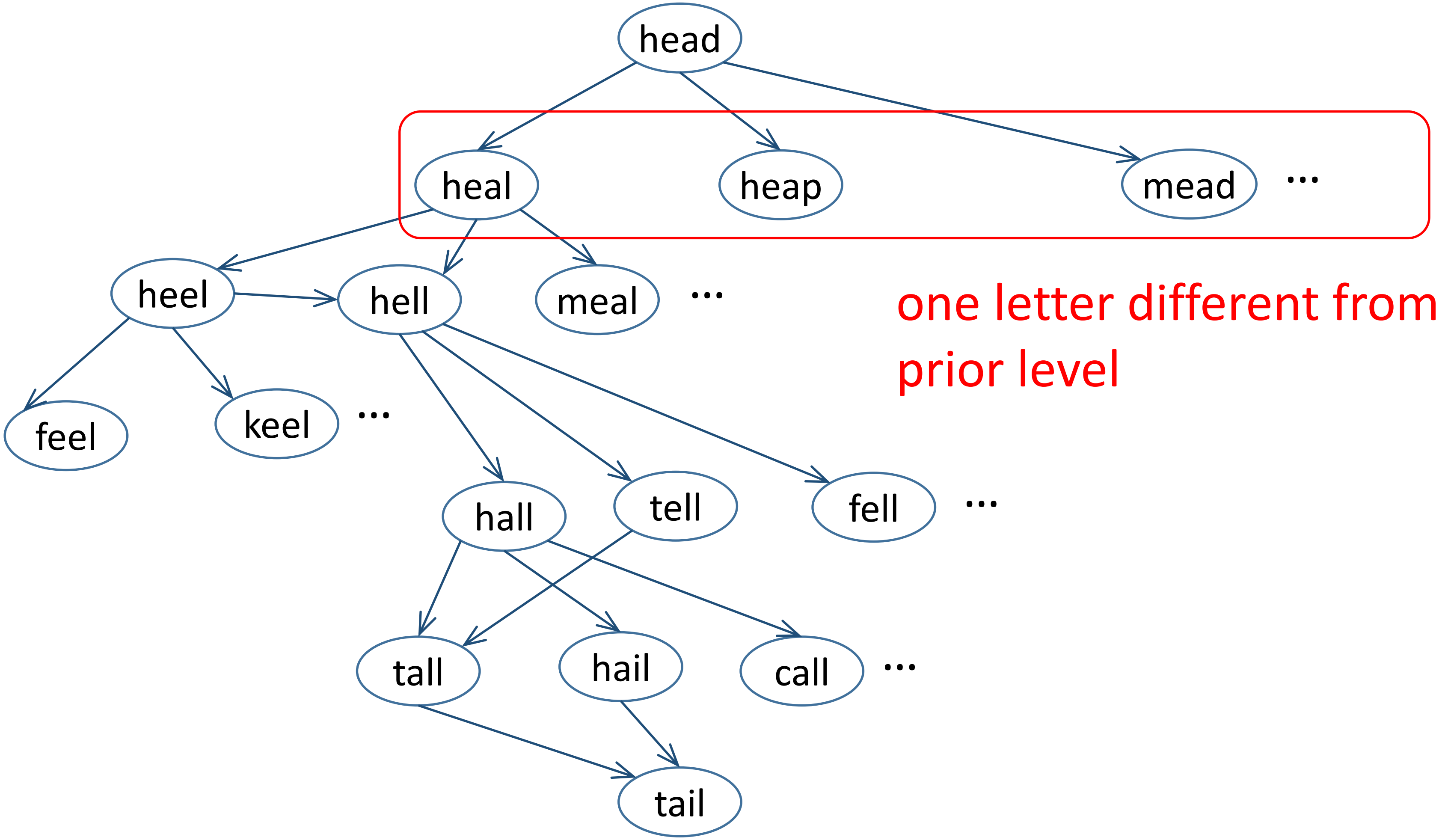
  - cat → cot → cog → dog

  head/tail

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

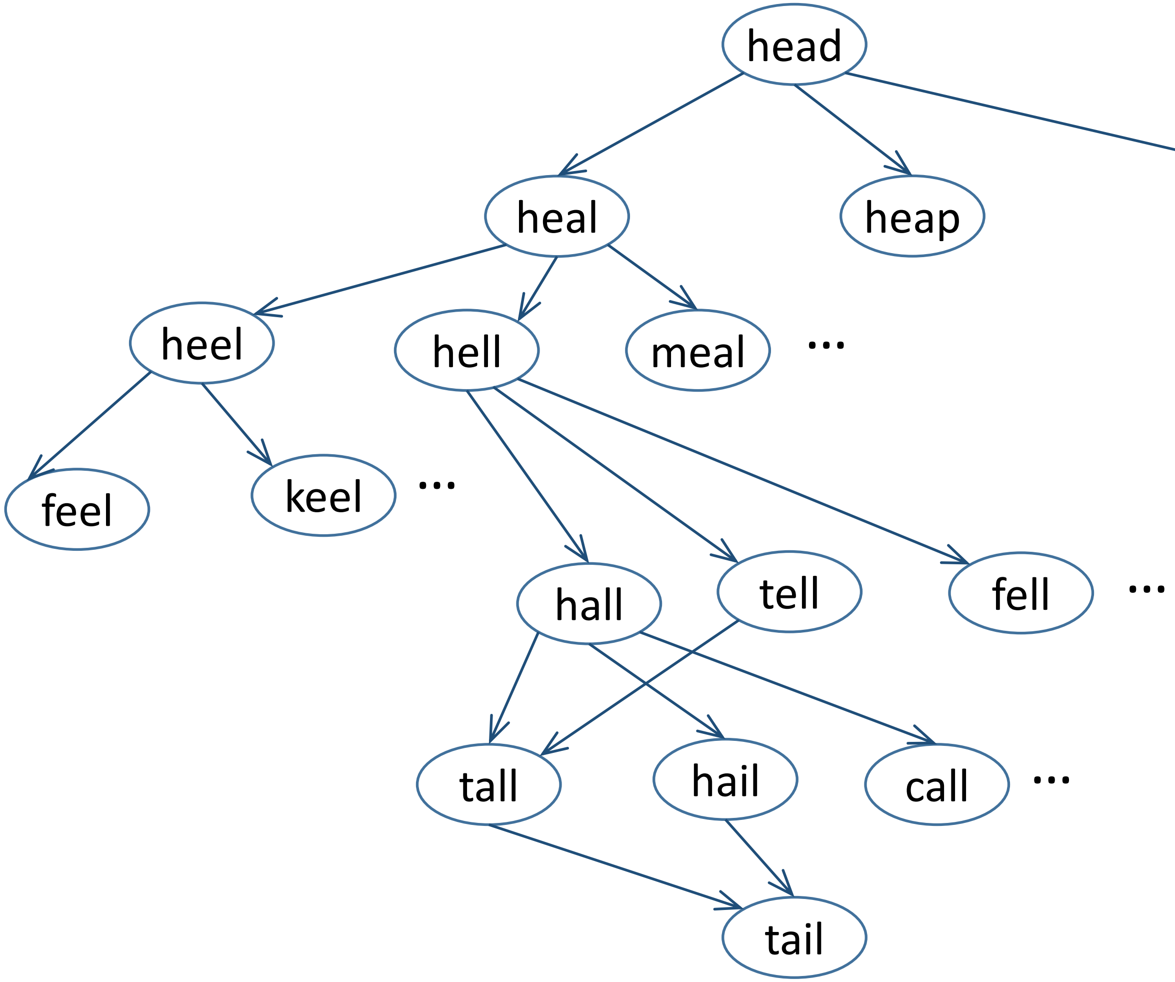
- Imagine a tree where each level is the set of possible words created by changing one character

Word morph: sample tree



one letter different from prior level

Word morph: sample tree



- How do we generate and search the possibilities?

Word morph

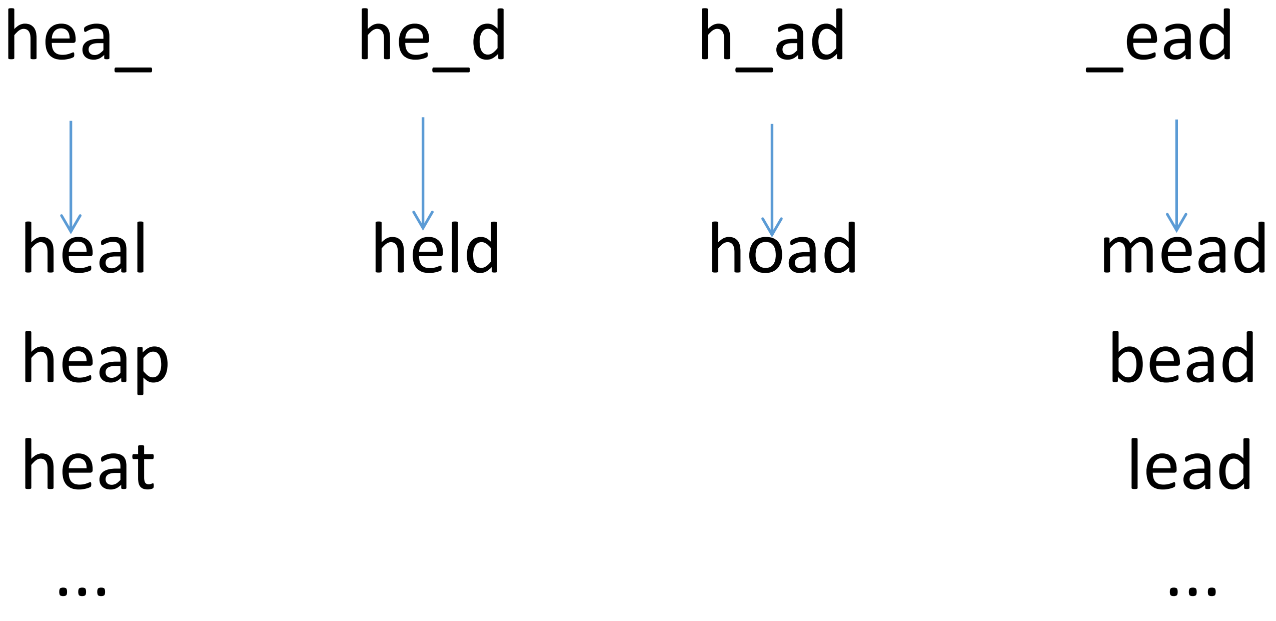
- Change one word into another by changing one letter at a time

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

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

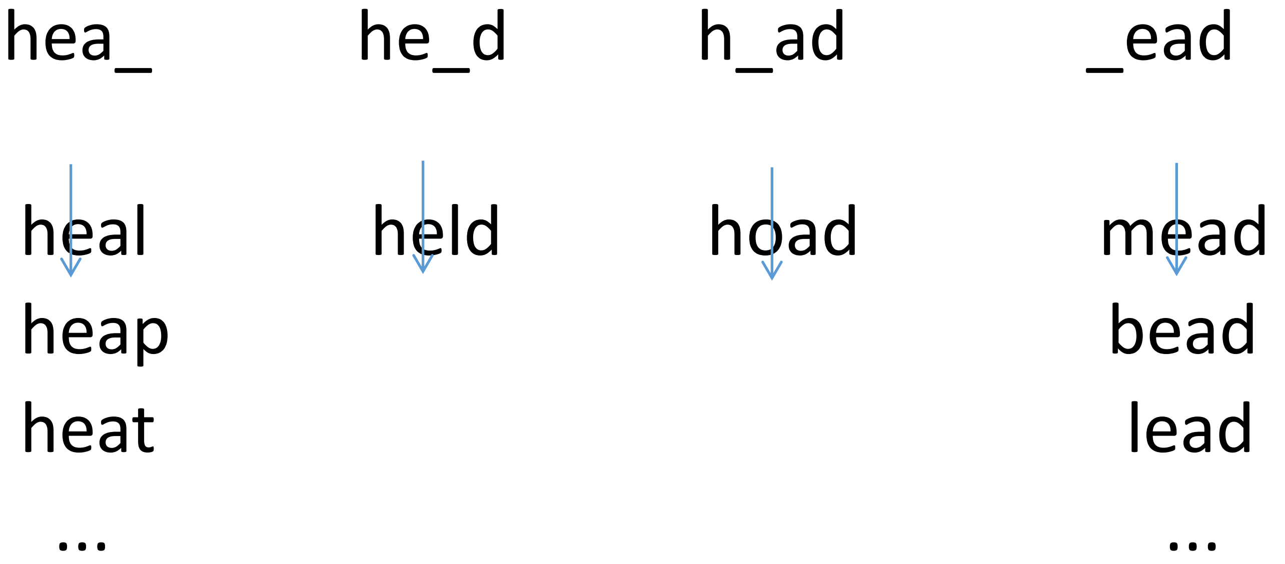
heap bead

heat lead

… …

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
heat			lead
…			…

Q: Where are the nonsense words (like heab)?

Word morph

- Given a list of valid words
  - Generate the set of words that differ from a word *w1* by one letter
  - Avoid adding words that are not valid
    - use a provided list of valid words to eliminate nonsense words

Exercise-ICA42, p. 1

Write a function next_words(w1, words_list) that takes a word, w1, and a list of valid words, words_list, and returns a list of words that differ from w1 by one letter.

Add only valid words that appears in words_list.

You can write Pseudocode.

Solution 1

- Given a list 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 let in the alphabet,
    create a new word by changing the letter at position i to let
    if the new word is in the list of valid words
        add it to the set of words*
```

*unless it's been seen already

Word morph

- Given a list of valid words
  - Generate the set of words that differ from a word *w1* 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 *w2* in the dictionary

    if the distance between *w1* and *w2* is 1, then add *w2* to the set of words*

  *unless it's been seen already

Exercise- ICA43-p.2

Write a function dist(w1, w2) that returns the number of positions where words w1 and w2 differer. 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. (Optional)

Word morph

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

Word morph: search

```
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 next  in candidate_list:
            #       Recurse with next candidate
            result = morph(next, w2, word_list, Seen + [w1])

            if result != []:
                return [w1] + result
        return []
```

generate the list of words to try next

Word morph: search

```
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 next  in candidate_list:
            #       Recurse with next candidate
            result = morph(next, w2, word_list, Seen + [w1])

            if result != []:
                return [w1] + result
        return []
```

Try the next “candidate”

Word morph: search

```
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 next in candidate_list:
            #    Recurse with next candidate
            result = morph(next, w2, word_list, Seen + [w1])

            if result != []:
                return [w1] + result
        return []
```

search from candidate

Word morph: search

```
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 next in candidate_list:
            #  Recurse with next candidate
            result = morph(next, w2, word_list, Seen +[w1])

            if result != []:
                return [w1] + result
        return []
```

if a solution is found, return immediately. Otherwise, keep searching (i.e., iterating).

Word morph: search

```
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:
            #  Recurse with next candidate
            result = morph(next, w2, word_list, Seen +[w1])

            if result != []:
                return [w1] + result
        return []
```

No solution found from the list of candidates;
“Backtrack” to prior level

Word morph: utility functions

```
def next_words(wd1, wd2, word_list):
    cands = [wd for wd in word_list \
                if len(wd) == len(wd1) and dist(wd, wd1) == 1]
    cands.sort()
    return cands

# 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)
```

Word morph: code

```
# 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: ' + dictfilename)
        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)

# dist(w1, w2) returns the no. of positions where w1, w2
differ.
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()
```

Word morph: example runs

- cat → dog
  - cat, cot, cog, dog
- head → tail
  - head, heal, ~~heel,~~ hell, hall, tall, tail
- nose → chin

  why the extra words? ☹

  dome

  - 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

Exercise- ICA42-p.3 & 4

Do the final exam review problems.

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

blank

e.g.: software → soft are → soft ear

- Looking for:

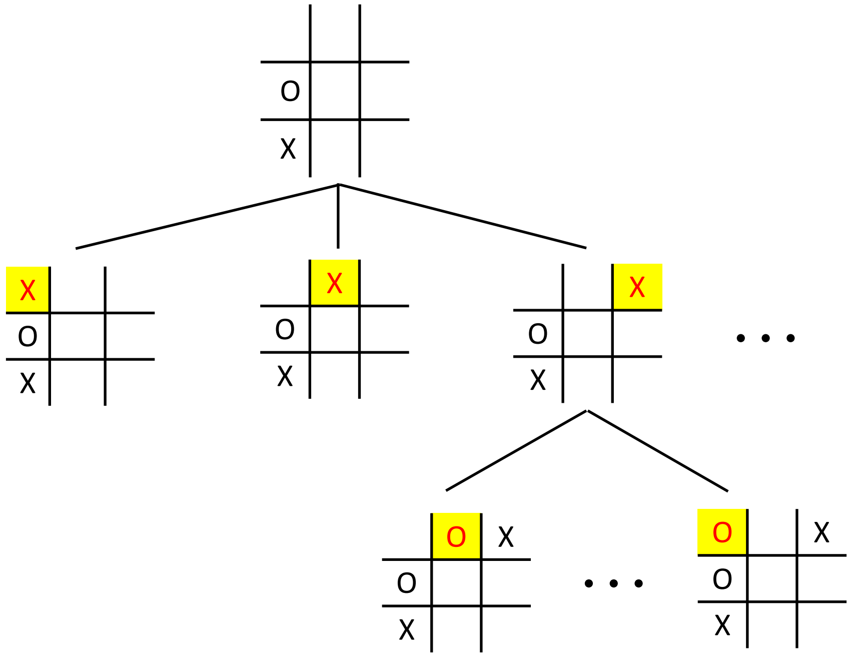
Wildcats → Beat ASU

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

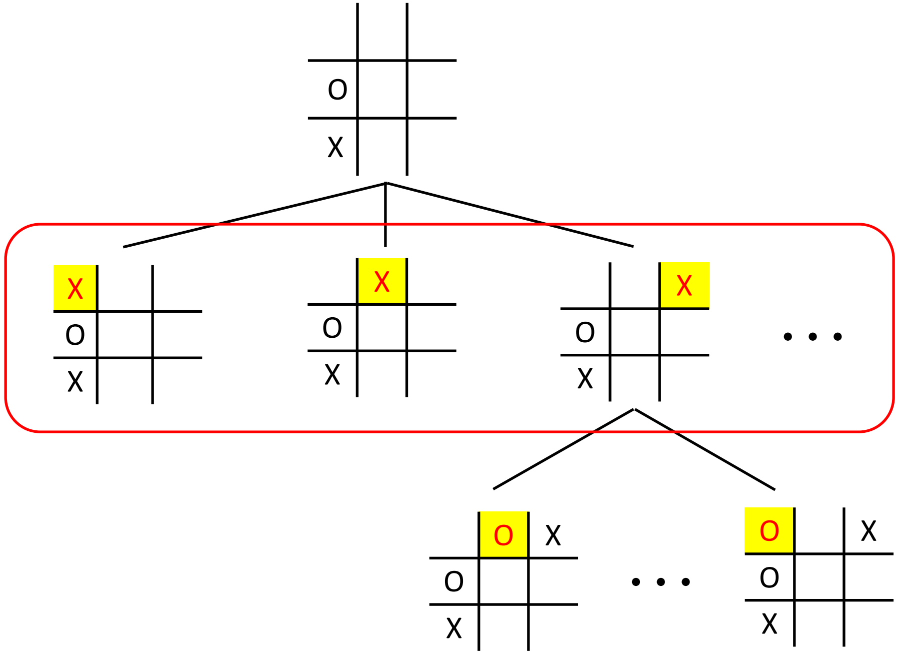


Game tree search revisited

Recall our tic-tac-toe program

Given a starting position,

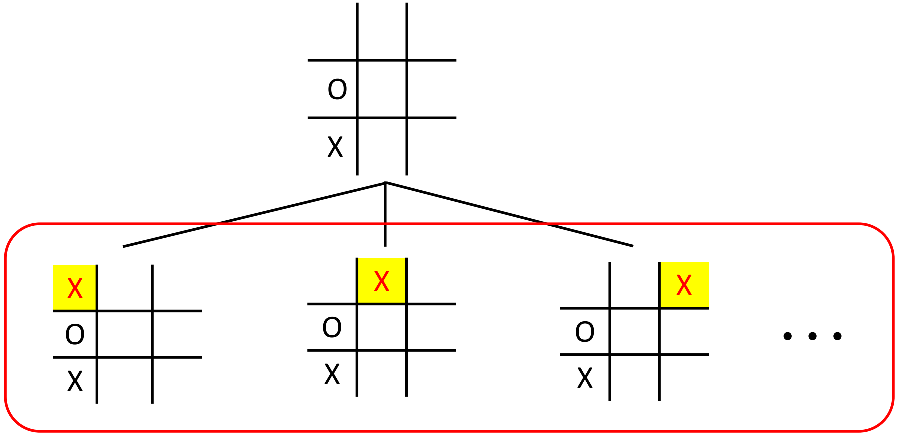
- it generates successive positions from different possible moves
- evaluates the effect of continuing play from each of these positions
- options
  - pick a move that leads to the best position after some number of turns n (n = "lookahead")
  - search exhaustively for a solution ←



Game tree search revisited

How do we generate the next possible position for x?

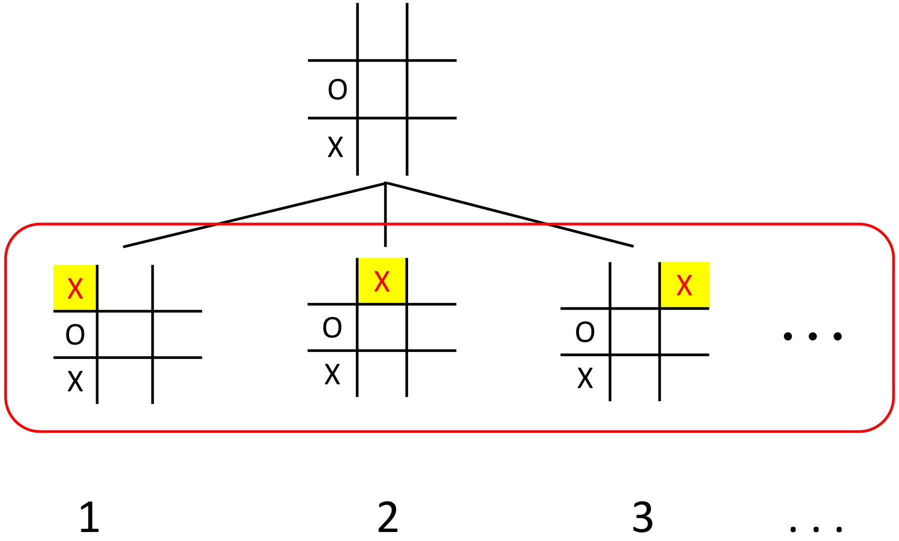
- iterate through the grid and find the next empty spot
- modify the board



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



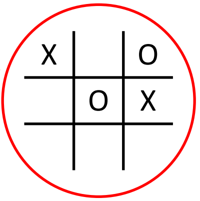
Game tree search revisited

```
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
 return "loss" indication
```

Game tree search revisited

```
def eval_pos(pos, turn):
    if game_over(pos):
        return win_or_loss(pos, turn)
    else: next_pos = pos
        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
```

'X' or 'O'



next = { 'X':'O', 'O':'X' }

Game tree search revisited

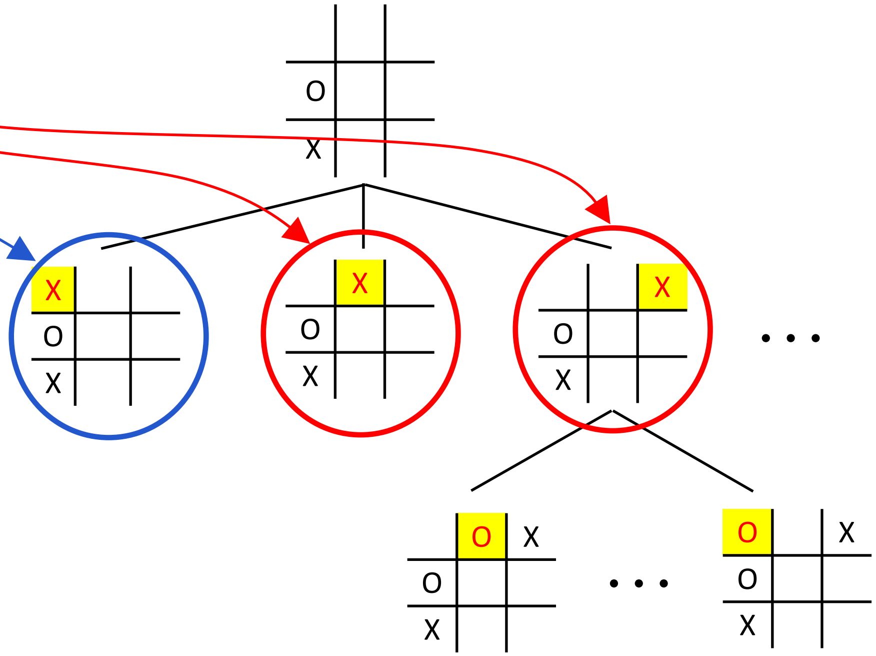
```
def eval_pos(pos, turn):
    if game_over(pos):
        return win_or_loss(pos, turn)
    else: next_pos = pos
        while next_pos != None:
            next_pos = generate_next_pos(pos, turn)
            if next_pos != None:
                result = eval_pos(next_pos, next[turn])
                ...
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
```

updates the position

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!

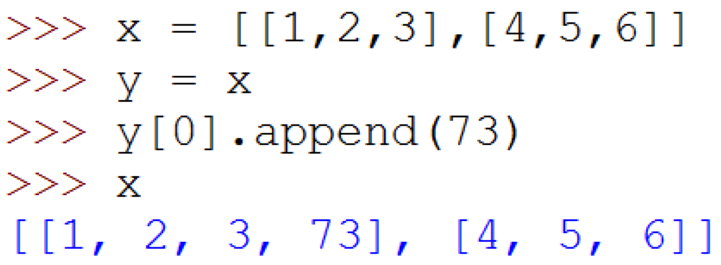
☹

Game tree search revisited

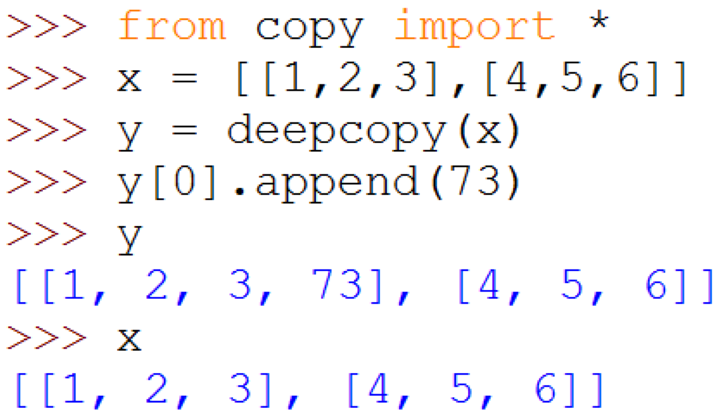
Solution: create a copy of the board position

A refresher on copying:

without copying	with deep copying



```
>>> 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

```
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