AD1 revision, the graph library, and graph search

Frej Knutar Lewander

Uppsala University
Sweden

31st October 2023

Outline

1. Python
2. Data Structures revision
3. Graphs
4. Breadth- and Depth-First Search
5. Greedy Algorithms

Why Python

The coding part of the assignments is done in Python. Why have we decided to use python, when

- other languages are faster (Java, C, C++, etc.),
- you haven’t used Python before,
- Python is not your language of choice, etc.

Why Python

The coding part of the assignments is done in Python. Why have we decided to use python, when

- other languages are faster (Java, C, C++, etc.),
- you haven’t used Python before,
- Python is not your language of choice, etc.

The ideas and mindset gained from this course are independent of the programming language they are applied to.

Python is just “a means to an end”.

Arrays

An array A (list in Python) of size n holds n elements:

A_i or $A[i]$ is the element at index i of A.

Arrays

An array A (list in Python) of size n holds n elements:

A_i or $A[i]$ is the element at index i of A.

Given an array A of size n, the average case time complexity for:

- accessing an element at index i ($0 \leq i < n$): $\Theta(?)$
- adding an element: $\Theta(?)$
- removing an element: $\Theta(?)$
- checking if A contains the element e: $\Theta(?)$
- adding an element at index n: $\Theta(?)$

Arrays

An array A (list in Python) of size n holds n elements:

A_i or $A[i]$ is the element at index i of A.

Given an array A of size n, the average case time complexity for:

- accessing an element at index i $(0 \leq i < n)$: $\Theta(1)$
- adding an element: $\Theta(n)$
- removing an element: $\Theta(n)$
- checking if A contains the element e: $\Theta(n)$
- adding an element at index n: $\Theta(1)$

Arrays

Arrays tend to be the default data structure of choice.
Here are some (non-exhaustive) tips on when to use an array:

- checking if an element exists in the array does not worsen the time complexity;
- the order of the elements matter; or
- the elements can be accessed by their (distinct) index fast (does not worsen the time complexity).

Sets

A set S of *cardinality* n holds n *distinct* elements:

A set is (typically) unordered.

Sets

A set S of *cardinality* n holds n *distinct* elements:

A set is (typically) unordered.

Given a set S of cardinality n, the average case time complexity for:

- checking if S contains the element e: $\Theta(?)$
- adding an element to S: $\Theta(?)$
- removing an element from S: $\Theta(?)$

Sets

A set S of *cardinality* n holds n *distinct* elements:

A set is (typically) unordered.

Given a set S of cardinality n, the average case time complexity for:

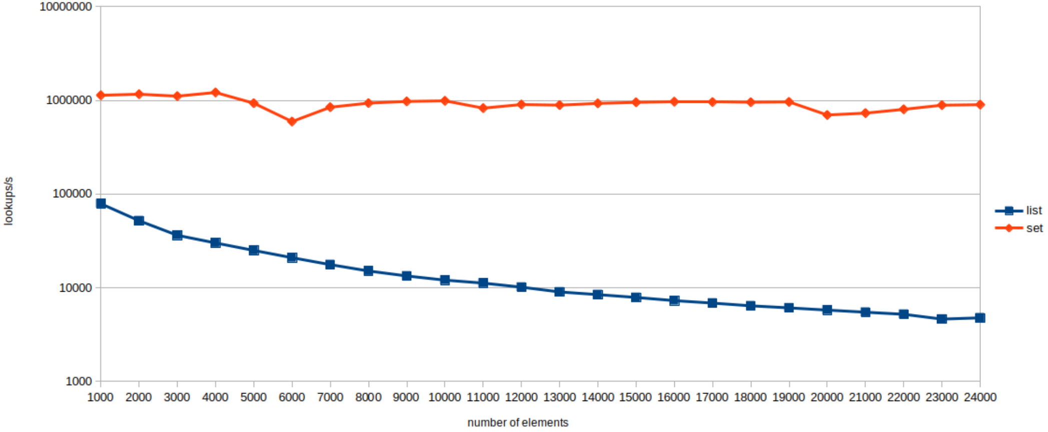
- checking if S contains the element e: $\Theta(1)$
- adding an element to S: $\Theta(1)$
- removing an element from S: $\Theta(1)$

Sets

Here are some (non-exhaustive) tips on when to use a set:

- checking if an element exists in the set is integral to the task at hand;
- duplicate elements are not allowed; or
- each element can *not* be accessed by a (distinct) index.

Comparison (logarithmic)



Hash Tables

A hash table H (dictionary in Python) of *cardinality* n holds n elements, each identified by a distinct key:

H_k or $H[k]$ is the element identified by the key k, where k is a key in H.

A hash table is (typically) unordered.

Hash Tables

A hash table H (dictionary in Python) of *cardinality* n holds n elements, each identified by a distinct key:

H_k or $H[k]$ is the element identified by the key k, where k is a key in H.

A hash table is (typically) unordered.

Given a hash table H with cardinality n, the average case time complexity for:

- accessing an element with key k: $\Theta(?)$
- checking if H contains the key k: $\Theta(?)$
- adding an element to H: $\Theta(?)$
- removing an element from H: $\Theta(?)$

Hash Tables

A hash table H (dictionary in Python) of *cardinality* n holds n elements, each identified by a distinct key:

H_k or $H[k]$ is the element identified by the key k, where k is a key in H.

A hash table is (typically) unordered.

Given a hash table H with cardinality n, the average case time complexity for:

- accessing an element with key k: $\Theta(1)$
- checking if H contains the key k: $\Theta(1)$
- adding an element to H: $\Theta(1)$
- removing an element from H: $\Theta(1)$

Hash tables

Hash tables tend to be the data structure of choice for simpler nested data. Here are some (non-exhaustive) tips on when to use a hash table:

- a set with some extra data is required or
- the data is simple, and creating classes (or other custom data structures) for the data is “overkill”.

Graphs

The graph.py Library (1)

A graph is represented similarly to the Adjacency list: the edges are represented as a dictionary with nodes as keys and sets of nodes as elements.

For a graph $G = (V, E)$:

Space complexity:	$\Theta(n + m)$
checking if G contains an edge (u, v):	$\Theta(1)$
Identifying all edges:	$\Theta(m)$

The graph.py Library (2)

Given a graph `G=(V,E)` the properties you need for the first assignment are:

- `G.edges` is a (duplicate free) unsorted list containing all edges `E` and
- `G.nodes` is a (duplicate free) unsorted list containing all vertices `V`

The graph.py Library (3)

Some graphs, $G = (V, E)$, have additional properties for the edges. In the graph library, an edge (u, v) can have:

- a *flow* that is integer or `None`, (typically) denoting the amount of some commodity that travels over (u, v);
- a *capacity* that is integer or `None`, (typically) denoting the maximum amount of some commodity that can travel over (u, v); and
- a *weight* that is integer or `None`, (typically) denoting the cost of including or traversing (u, v).

The graph.py Library (4)

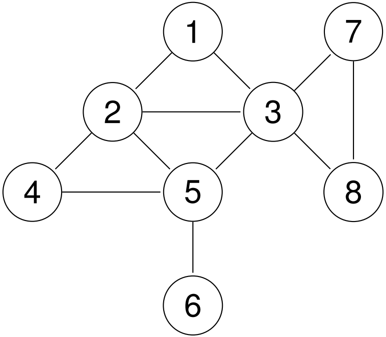
Given a graph `G` and an edge `(u,v)`, the flow, capacity, and weight of `(u,v)` be accessed or modified by the methods:

- `G.flow(u, v)` and `G.set_flow(u, v, f)`;
- `G.capacity(u, v)` and `G.set_capacity(u, v, c)`; and
- `G.weight(u, v)` and `G.set_weight(u, v, w)`,

respectively for some $\mathtt{f}, \mathtt{c}, \mathtt{w} \in \{\mathtt{None}\} \cup \mathbb{Z}$.

Breadth- and Depth-First Search

Given an unordered graph $G = (V, E)$, starting at node s, is there a path in G from s to t?



BFS – pseudocode

```
algorithm BFS($G, s, t$)
    $Visited \leftarrow \{s\}$
    $Q \leftarrow$ an empty queue
    ENQUEUE($Q, s$)
    while $|Q| > 0$ do                                   // Variant: ...
        $u \leftarrow$ DEQUEUE($Q$)
        if $u = t$ then
            return true
        for each $(w, v) \in E$ where $w = u$ do         // Variant: ...
            if $v \notin Visited$ then
                ENQUEUE($Q, v$)
                $Visited \leftarrow Visited \cup \{v\}$
    return false
```

BFS – pseudocode

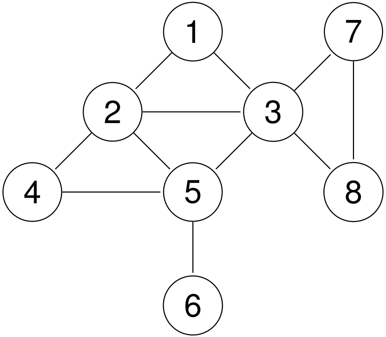
```
algorithm BFS($G, s, t$)
    $Visited \leftarrow \{s\}$
    $Q \leftarrow$ an empty queue
    ENQUEUE($Q, s$)
    while $|Q| > 0$ do                                   // Variant: $n - |Visited|$
        $u \leftarrow$ DEQUEUE($Q$)
        if $u = t$ then
            return true
        for each $(w, v) \in E$ where $w = u$ do          // Variant: $degree(u) - \#$iterations
            if $v \notin Visited$ then
                ENQUEUE($Q, v$)
                $Visited \leftarrow Visited \cup \{v\}$
    return false
```

BFS – Python code

```python
def bfs(graph: Graph, s:str, t:str) -> bool:
    visited = {s}
    queue = deque()
    queue.append(s)
    while len(queue) > 0:
        u = queue.popleft()
        if u == t:
            return True
        for v in graph.neighbors(u):
            if v not in visited:
                queue.append(v)
                visited.add(v)
    return False
```

BFS

With $s = 1$ and $t = 6$, what set of nodes will be visited during $\textsc{Bfs}(G, s, t)$? Can the set differ between runs?



DFS – pseudocode

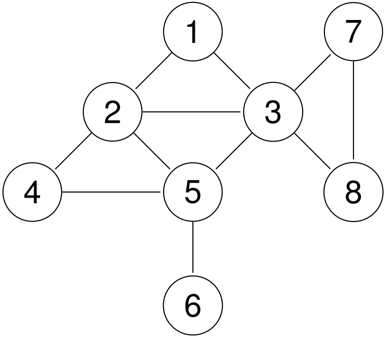
```
**algorithm** DFS($G, s, t$)
    $Visited \leftarrow \{s\}$
    $S \leftarrow$ an empty stack
    PUSH($S, s$)
    **while** $|S| > 0$ **do**                                   // Variant: $n - |Visited|$
        $u \leftarrow$ POP($S$)
        **if** $u = t$ **then**
            **return true**
        **for each** $(w, v) \in E$ **where** $w = u$ **do**      // Variant: $degree(u) - \#$iterations
            **if** $v \notin Visited$ **then**
                PUSH($S, v$)
                $Visited \leftarrow Visited \cup \{v\}$
    **return false**
```

DFS – pseudocode

```
algorithm DFS($G, s, t$)
    $Visited \leftarrow \{s\}$
    $S \leftarrow$ an empty stack
    PUSH($S, s$)
    while $|S| > 0$ do                                          // Variant: $n - |Visited|$
        $u \leftarrow$ POP($S$)
        if $u = t$ then
            return true
        for each $(w, v) \in E$ where $w = u$ do                // Variant: $degree(u) - \#$iterations
            if $v \notin Visited$ then
                PUSH($S, v$)
                $Visited \leftarrow Visited \cup \{v\}$
    return false
```

DFS

With $s = 1$ and $t = 6$, what set of nodes will be visited during $\textsc{Dfs}(G, s, t)$? Can the set differ between runs?



DFS – Python code

```python
def dfs(graph: Graph, s:str, t:str) -> bool:
  visited = {s}
  stack = []
  stack.append(s)
  while len(stack) > 0:
    u = stack.pop()
    if u == t:
      return True
    for v in graph.neighbors(u):
      if v not in visited:
        stack.append(v)
        visited.add(v)
  return False
```

DFS – Python code (contd)

Why can we use a list as a stack without worsening the time complexity?

DFS – Python code (contd)

Why can we use a list as a stack without worsening the time complexity?

Adding an element to the end of a list or array has (amortised) time complexity $\Theta(1)$. We discuss amortised time complexity in AD3.

We could also initialize a list with *n* elements and keep a “stack pointer”:

DFS with Stack Pointer – Python code

```python
def dfs_sp(graph: Graph, s:str, t:str) -> bool:
  visited = {s}
  stack = [s] + ([None] * (len(graph.nodes) - 1))
  top = 0 # stack[top] = top element of stack
  while top >= 0:
    u = stack[top]
    top -= 1
    if u == t:
      return True
    for v in graph.neighbors(u):
      if v not in visited:
        top += 1
        stack[top] = v
        visited.add(v)
  return False
```

Greedy Algorithms

A greedy algorithm makes the locally optimal choice in every step.

Greedy Algorithms - Coin Change

We are to return the (cash) change 19 SEK using the fewest amount of coins (coin denominations are 10 SEK, 5 SEK, 2 SEK, and 1 SEK).

Greedy Algorithms - Coin Change

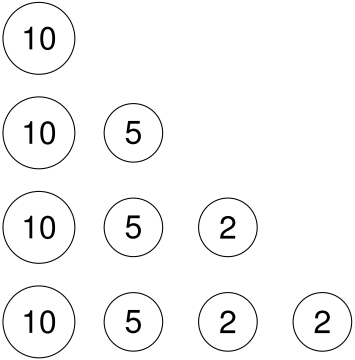
We are to return the (cash) change 19 SEK using the fewest amount of coins (coin denominations are 10 SEK, 5 SEK, 2 SEK, and 1 SEK).

$$19 - 10 = 9$$

$$9 - 5 = 4$$

$$4 - 2 = 2$$

$$2 - 2 = 0$$



Greedy Algorithms

When are greedy algorithms good?

Greedy Algorithms

When are greedy algorithms good?

When selecting the locally optimal choice in every step yields a sufficiently good global solution.

In the coin example, the global solution was optimal, but that is not guaranteed.