Algorithms&Complexity - In Class Assignments (Week 4)

Randomized and Online Algorithms

Manuela Cleves

1. Karger's min-cut Algorithm Input: Connected, undirected, unweighted graph $G=(V,E)$.

Output: Smallest list of edges, which if removed will cut G in two.

Algorithm: 1) Select a random edge. 2) Remove this edge and merge its nodes into one node. 3) Remove any cycles, if present 4) Repeat 1-3 until only one edge remains.

Note: This is a Monte-Carlo algorithm, so...?

Notes on Monte-Carlo algorithms:

- Algorithms of this type can output a wrong answer.
- They have a bounded runtime.
- By running this algorithm multiple times, we can decrease the chance of an incorrect solution.
- If our algorithm is of 1-side error type the following relation is true: $(1-p)^t \leq e^(-pt)$.
- If we ran our algorithm 10 times, we would get an error probability of $e^(-10)$.

In [ ]:
```
import random

def min_cut(graph):
    while len(graph) > 1:
        # Select a random edge - randomness because it's monte carlo
        edge = random.choice([(node, neighbor) for node in graph for neighbor :

        # Remove that edge
        n1, n2 = edge
        if n2 not in graph:
            continue  # Skip this iteration if n2 is not in the graph
        graph[n1].extend(graph[n2])
        del graph[n2]

        # Merge nodes and remove cycles (step 3)
        graph[n1] = list(set(graph[n1]))
        graph[n1] = [node for node in graph[n1] if node != n1 and node not in g

    return list(graph.keys())[0]

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C', 'D'],
    'C': ['A', 'B', 'D'],
```

```
    'D': ['B', 'C']
}

min_cut_result = min_cut(graph)


#Because it's monte carlo, we might not get the correct answer! - should I run
print("Smallest list of edges to cut the graph:", min_cut_result)
```

2. Input: Randomly generate integers from 0 to 9, as inputs.

Implement FIFO and LRU algorithms. Test them for 1000 inputs and allow the user to choose their desired cache size. Track the process of your algorithm by copying the state of your cache into a list.

In [4]:
```
def fifo_cache(inputs, cache_size):
    cache = []
    cache_states = []

    # Of all inputs in the cache array, remove the oldest element (First "out",
    for input_value in inputs:
        if input_value not in cache:
            if len(cache) == cache_size:
                cache.pop(0)
            cache.append(input_value)
        # Save the state of the cache
        cache_states.append(list(cache))

    return cache_states

def lru_cache(inputs, cache_size):
    cache = []
    cache_states = []

    #Of all inputs, we will remove those already in cache
    for input_value in inputs:
        if input_value in cache:
            cache.remove(input_value)
        # We will also remove least recently used
        elif len(cache) == cache_size:
            cache.pop(0)
        cache.append(input_value)
        # Save the state of the cache
        cache_states.append(list(cache))

    return cache_states

# Randomly generate 1000 inputs
inputs = [random.randint(0, 9) for _ in range(1000)]

# Allow the choosing the cache size
cache_size = int(input("Enter the cache size: "))

# Run FIFO algorithm and track cache states
fifo_states = fifo_cache(inputs, cache_size)

# Run LRU algorithm and track cache states
lru_states = lru_cache(inputs, cache_size)
```

```
# Display the final cache states
print("FIFO Cache States:")
for state in fifo_states[-5:]:  # Display the last 5 states for brevity
    print(state)

print("\nLRU Cache States:")
for state in lru_states[-5:]:  # Display the last 5 states for brevity
    print(state)
```

```
FIFO Cache States:
[5, 6, 8, 2, 4, 0, 3, 9, 7, 1]
[5, 6, 8, 2, 4, 0, 3, 9, 7, 1]
[5, 6, 8, 2, 4, 0, 3, 9, 7, 1]
[5, 6, 8, 2, 4, 0, 3, 9, 7, 1]
[5, 6, 8, 2, 4, 0, 3, 9, 7, 1]

LRU Cache States:
[1, 7, 2, 9, 8, 0, 4, 5, 6, 3]
[1, 7, 2, 9, 8, 4, 5, 6, 3, 0]
[1, 7, 2, 9, 8, 4, 5, 3, 0, 6]
[1, 7, 2, 9, 8, 4, 5, 3, 6, 0]
[1, 7, 2, 9, 8, 5, 3, 6, 0, 4]
```

In [ ]:

Other code used in class with Ilia (NOT for grading):

In [ ]:

```
import numpy as np

#Create matrices
A = [[1,2],[3,4]]
B = [[1,2],[3,4]]
#C = [[1,3], [2,5]]

r =[]

# Generate a random column vector of size N*1, the first 2 is telling me that 
N = len(A)
print (result)

#Create function that multiplies matrixes

def multimatrix(A,B):
    result = []
    for i in range(len(A)):

        row = []
        for j in range(len(B[0])):

            product = 0
            for v in range(len(A[i])):
                product += A[i][v] * B[v][j]
            row.append(product)

        result.append(row)
    return result
```

In [ ]:
```
# C = multimatrix(A,B)
C = [[0,0], [0,0]]
```

In [ ]:
```
C_false = [[1,2],[3,4]]
```

In [ ]:
```
#Check if this is correct (#Calculate P=Ax(B*r)-(C*r), you repeat this many ti

for i in range(10):
   R = np.random.randint(2, size=(N, 1))
   Step2=multimatrix(B,R)
   Step3=multimatrix(C,R)
   Step1=multimatrix(A,Step2)
   P=np.array(Step1)-np.array(Step3)

   if sum(P)==0:
      print(True)
   else:
      print(False)
```

In [ ]:
```
Step3
```

In [ ]:
```
C
```

In [ ]:
```

```

In [ ]:
```
#Randomly shuffle values in an array until the correct order is achieve (shuff
```

In [ ]:
```
#Create an array - this you did worng, you were maybe supposed to sort it (so
#But that is not efficient, so quick sort is better

arr=[[3],[4],[1],[2],[3]]
copyarr=[[3],[4],[1],[2],[3]]
#Shuffle the array
import random
random.shuffle(arr)
print(arr)

while arr != copyarr:
    random.shuffle(arr)
    print(arr)
```

In [ ]:
```
#This is a deterministic quicksort, so now try to shuffle it (it is always cor
#FIX, BEST TO NOT USE TEMPLATE CODE, you would have to track all the smallest
#That would be the split, so we call quicksort on the smallest, and the larges

def quicksort(arr):
    if len(arr) == 1 or len(arr) == 0:
        return arr
    else:
        pivot = arr[0]
        i = 0
        for j in range(len(arr)-1):
            if arr[j+1] < pivot:
                arr[j+1], arr[i+1] = arr[i+1], arr[j+1]
                i += 1
                print(arr)
```

```
        arr[0], arr[i] = arr[i], arr[0]#restructure arr to have allthesamllest
        first_part = quicksort(arr[:i])#all the smallest values to the left (b
        second_part = quicksort(arr[i+1:])
        first_part.append(arr[i])
        return first_part + second_part

alist = [54,26,93,17,77,31,44,55,20]
print(quicksort(alist))
```

In [ ]: