Backtracking Again

Thomas Schwarz, SJ

Generic Backtracking

- Backtracking improves on Brute-Force Enumeration
  - Where solution is can be generated recursively
- Gain traction by avoiding generating potential solutions if we can determine that they will not work

Generic Backtracking

- Three conditions
  - Solve by iteratively generating partial solutions
  - Determine whether a partial solution is invalid
    - Otherwise, it will just be complete enumeration
  - Determine whether a solution is complete

Generic Backtracking

- Generic solution: Uses recursive calls and thereby OS stack
  - Need to fill in the problems

```
def solver(board):
    If complete return board
    Find next step
    For all possible ways to take the next step:
        take step this way
        if we are in a valid partial state:
            result = solver(board)
            if result is complete: return result
        undo the step
    Return board
```

Gray Codes

- Ordering of all integers of certain bit-length
  - Each number differs from predecessor by one bit
  - Example length three:

```
0: 000
1: 001
3: 011
2: 010
6: 110
7: 111
5: 101
4: 100
```

Gray Codes

- Implement Gray codes as a list, starting with [0]
- To implement scheme
  - Need to:
    - Find next possible element:
      - Flipping a bit: Exclusive-Oring with a power of two
      - Example
        - 18 = b10010
        - 18^8 = b11010 = 26

Gray Codes

- To implement scheme:
  - Find all choices:
    - ```
      for b in [1,2,4,8,16]:
      x = lista[-1]^b
      ```
      ```

Gray Codes

- Implementing scheme
  - Determine whether a solution is feasible:
    - Check that list contains different elements
    - Only need to check the last element since the previous ones already fulfill
      - `if x not in lista`

Gray Codes

- To implement scheme:
  - Determine when we are done:
    - By construction:
      - subsequent integers differ in one bit
      - all integers are different
- Therefore:
  - Do we have 2^n elements?

```
if len(lista) == 16:
```

Gray Codes

- Build a new partial solution from previous partial solution and a choice?
  - Choice is a new number that differs in one bit from last element in list
  - Append new number to list

```
for b in [1,2,4,8]:
   x = lista[-1]^b
   if x not in lista:
      lista.append(x)
```

Gray Codes

- Undo an unsuccessful expansion:
  - Just remove recently appended number from list
  - Easiest with pop
  -
```
lista.pop()
```

Gray Codes

- Putting things together:

```
def solver(lista):
    if len(lista)==1024: #done?
        return lista
    for b in [1,2,4,8,16,32,64,128,256,512]:
      #get all possible choices
        x = lista[-1]^b
        if x not in lista:          #is feasible?
            lista.append(x)         #get new partial
            result = solver(lista) #does it work?
            if result and len(result)==1024:
                return result   #worked, we are done
            lista.pop()         #did not work, try next
```