Criterion C - Development

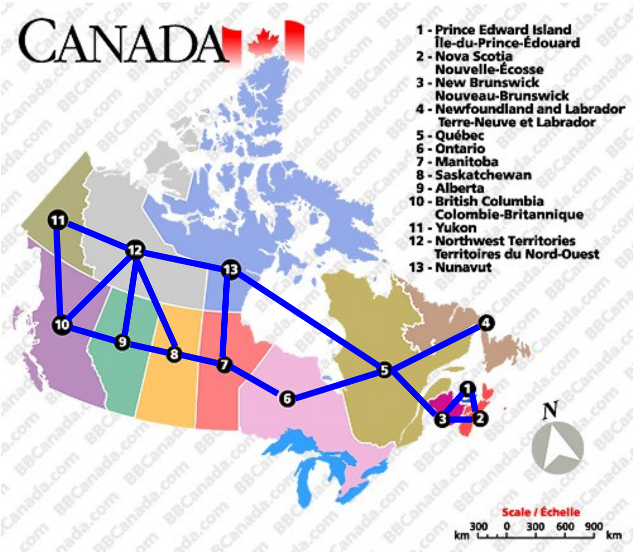
The product is a Python program. It accepts map data about bordering regions from a text-file written by the user. It automatically searches for a set of acceptable colors for the regions, using at most 4 colors, so that bordering countries never share the same color.

-List of techniques –

- Parameter passing
- Random number generating
- Indexing from one list to another

- Map Diagrams -

Computer scientists define a data-structure called a "map diagram". This records only the **connections** between regions, without recording long borders and large regions. This turns complex regions into simple connecting lines, like these blue lines:



Source: BBCanada http://www.bbcanada.com/bb_canada_map.cfm accessed 18 Feb 2011
'Maps courtesy of BBCanada.com'

Each line connects two neighboring regions. Notice that 4 regions in the middle meet at one point , but diagonal neighbors are NOT connected - e.g. #8 and #13 may have the same color. Each region only needs a list of neighbors, and that list is only as long as the number of line segments connecting to that region, like this:

```
YU --> BC,NW BC
--> YU,NW,AL
NW --> YU,NU,BC,AL,SA
```

```
 AL --> BC,NW,SA
SA   -->  AL,NW,MA
NU   -->  NW,MA,QU
MA --> SA,NU,ON
ON --> MA,QU
QU --> ON,NU,NF,NB
NF --> QU
NB --> QU,NS,PE
NS --> NB,PE PE
--> NB,NS
```

This requires minimal data entry. It's also easy because each entry only requires looking at a small part of the map, examining just the neighbors around a region. The order of the entries doesn't matter, so changes are easy. We''d like to just put a paper map into a scanner and let the computer figure out the borders - but that requires image-recognition and AI techniques well beyond my programming skills.

- Input/Output -

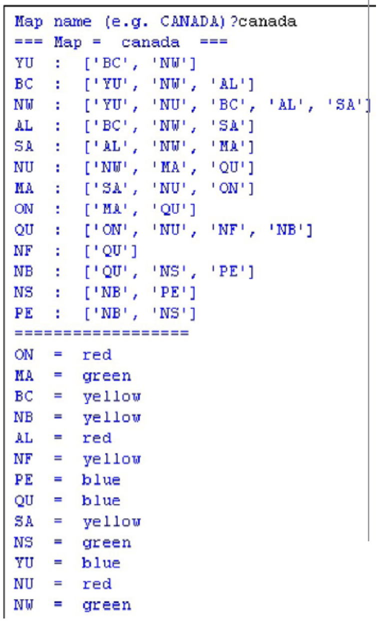
The user types the border data with a text editor (e.g. Notepad) and saves it in a text-file.
This data is for the Canada map (above). The first entry in each line specifies a region, while the further entries in the line are the neighbors of that region.
The program reads this file and stores the data in a convenient format in a **dictionary**.
→ *See code for the* **readBorders** *method.*

When a set of acceptable colors is found, the regions and matching colors are printed in a simple list as shown at the right.
→ *See code for the* **listColors** *method.*

Canada text-file

```
YU,BC,NW BC,YU,NW,AL
NW,YU,NU,BC,AL,SA
AL,BC,NW,SA
SA,AL,NW,MA
NU,NW,MA,QU
MA,SA,NU,ON ON,MA,QU
QU,ON,NU,NF,NB NF,QU
NB,QU,NS,PE
NS,NB,PE
PE,NB,NS
```

Sample Output



```
Map name (e.g. CANADA)?canada
=== Map =  canada  ===
YU  :  ['BC', 'NW']
BC  :  ['YU', 'NW', 'AL']
NW  :  ['YU', 'NU', 'BC', 'AL', 'SA']
AL  :  ['BC', 'NW', 'SA']
SA  :  ['AL', 'NW', 'MA']
NU  :  ['NW', 'MA', 'QU']
MA  :  ['SA', 'NU', 'ON']
ON  :  ['MA', 'QU']
QU  :  ['ON', 'NU', 'NF', 'NB']
NF  :  ['QU']
NB  :  ['QU', 'NS', 'PE']
NS  :  ['NB', 'PE']
PE  :  ['NB', 'NS']
==================
ON  =  red
MA  =  green
BC  =  yellow
NB  =  yellow
AL  =  red
NF  =  yellow
PE  =  blue
QU  =  blue
SA  =  yellow
NS  =  green
YU  =  blue
NU  =  red
NW  =  green
```

- Algorithms to Search for a Color Scheme -

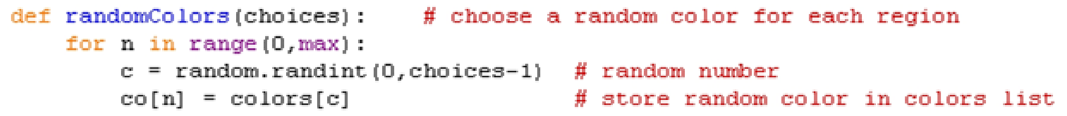
There are two possible strategies for searching for creating the map using the acceptable colors:

1. Search for an algorithm that always produces a successful coloring scheme

- OR -

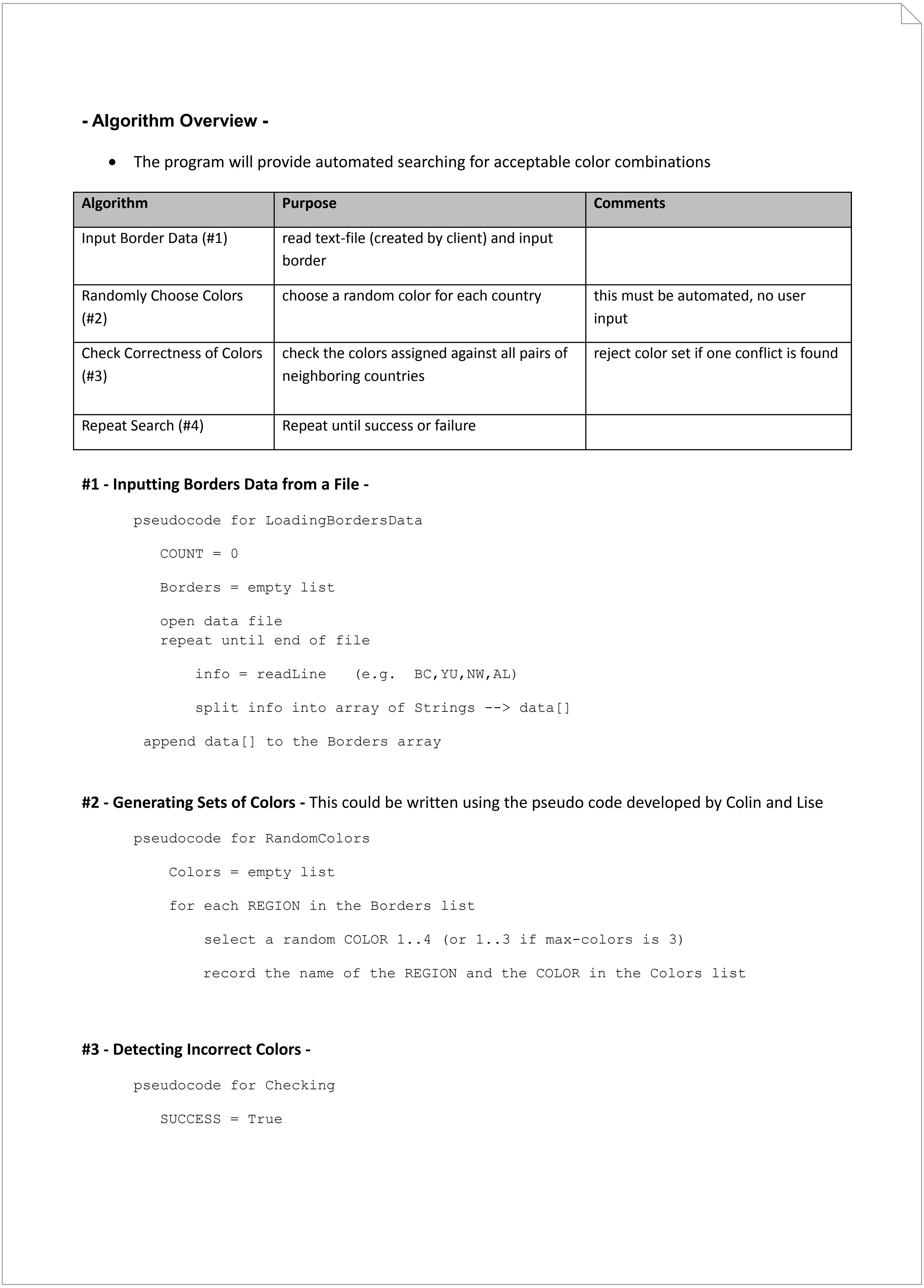
2. assign colors randomly and check whether the set of colors is acceptable (neighbors don't have the same color) - if the colors don't work, then repeat with a different random set, until a successful set is found (or quit after 1000 tries)

I was unable to find a straightforward algorithm that works for every map, therefore the solution I developed uses a random guess and check strategy. → *See code for the* **randomColors** *method.*



```
def randomColors(choices):     # choose a random color for each region
    for n in range(0,max):
        c = random.randint(0,choices-1)   # random number
        co[n] = colors[c]                 # store random color in colors list
```

The program tries 1000 times to find a successful color scheme (it may be necessary to re-run another 1000, several times, to succeed with a difficult map). For each color scheme, it must check whether the colors are acceptable. → *See source code for the* **checkColors** *method.*



- Algorithm Overview -

- The program will provide automated searching for acceptable color combinations

Algorithm	Purpose	Comments
Input Border Data (#1)	read text-file (created by client) and input border	
Randomly Choose Colors (#2)	choose a random color for each country	this must be automated, no user input
Check Correctness of Colors (#3)	check the colors assigned against all pairs of neighboring countries	reject color set if one conflict is found
Repeat Search (#4)	Repeat until success or failure	

#1 - Inputting Borders Data from a File -

```
pseudocode for LoadingBordersData

   COUNT = 0

   Borders = empty list

   open data file
   repeat until end of file

       info = readLine   (e.g.  BC,YU,NW,AL)

       split info into array of Strings --> data[]

 append data[] to the Borders array
```

#2 - Generating Sets of Colors - This could be written using the pseudo code developed by Colin and Lise

```
pseudocode for RandomColors

    Colors = empty list

    for each REGION in the Borders list

        select a random COLOR 1..4 (or 1..3 if max-colors is 3)

        record the name of the REGION and the COLOR in the Colors list
```

#3 - Detecting Incorrect Colors -

```
pseudocode for Checking

   SUCCESS = True
```

```
for each REGION in the Borders list

    for each NEIGHBOR of the REGION

        look up REGION.COLOR in the Colors list
        look up NEIGHBOR.COLOR in the Colors list

        if REGION.COLOR == NEIGHBOR.COLOR

             SUCCESS = False

return SUCCESS
```

#4 – Repeat search -

Show how program shows whether search has been successful or not.

- Data Structures -

The program uses several types of lists:

- **tuples** - standard names of 4 colors are stored in a tuple (which never changes): `colors = ("red","green","blue","yellow")`
- **arrays -** the list of randomly chosen colors is stored in an array, indexed by integers :

```
def  randomColors()  :
    for  nin  range(0,max)  :
        c =  random.randint(0,choices-1)
        co[n]  =  colors[c]
```

- **dictionaries**

  - the **borders** data is stored in a dictionary :

```
border  =  {"YU":["BC","NW"],
           "BC":["YU","NW","AL"],
           "NW":["YU","NU","BC","AL","SA"],
           .....
          }
```

This makes it easy to store the neighbors together with a region, as well as making it easy to retrieve data by name (no search method required).

- the **state** dictionary contains each country code and a corresponding number:

```
state  =  {"YU":0,
           "BC":1,
           "NW":2,
            ...
          }
```

The purpose of the **state** dictionary is to convert a country code into an index number that is used to get the matching color in the color array. This is more convenient than a search method.

These lists are used to combine color assignments and border data in the checkingColors method -
→ *see source code for the* **checkColors** *method.*

- Program Code Listing -

The program is written with good style and reusable methods with parameters and return values, improving readability and maintainability. The **dicitionary** structure and convenient Python commands like **slice** make the program relatively short.

** Words = 300 **