TDT4136 Logic and Reasoning Systems

Jørgen Grimnes
Assignment 4

Fall 2013

1 Simulated Annealing

The search node

Since Python code is pretty self-explanatory, I have appended a few lines of comment to the script instead of writing an extended explanation here. These extended comments are found in the source script.

```
class Node:
    def __init__(self, gene):
        # The node must contain a cost value
        self.gene = gene
        self.cost = self.objective_function( gene )

    def random_neighbour(self):
        # **IMPLEMENT**
        # generate neighbors
        """
        We only care about pseudo generating the neighbors,
        since we are going to pick just one of them. Thus
        we only need to perform a "small" random alteration
        to the parent.
        """
        return neighbor

    def objective_function(self, gene ):
        # **IMPLEMENT**
        # evaluate this state

        return  evaluation
```

Figure 1: The python implementation of the generalized search node

Handling the temperature variations

```
def initial_temp( node ):
    """
    f.ex upper bound on (max cost - min cost)

    The example below belongs to the Egg Box exercise.
        Returns the maximum cost possible by putting an egg in each
        of the availible spots.
    """
    tmp = np.ones(shape=node.gene.shape,dtype="uint8")
    return float(node.objective_function( tmp ))

def dependant_initial_t( temperature,node,neighbor ):
    """
    Increase the temperature to make this neighbor a very likely choice.
    This helps us to "correct" the initial temperature if it was initially
    too low.
    This function may not decrease the temperature.
    """
    x = temperature/(abs(node.cost-neighbor.cost)+1)*math.log1p(100/99)
    return temperature/x if 1>x>0 else temperature

def reduce_t( temperature,repetitions ):
    """
    common function for temperatur reduction
    """
    return temperature/math.log(repetitions+2,2)
```

Figure 2: How this script handles the temperature variations.

The main search function

```
def sa( start_node, goal_cost ):
    """

    d : # of repetitions pr temperature value
        This should initially be set to something like the size( neighborhood )
        but since we're only generating a single neighbor, we're forced to
        approximate.

    """
    node = start_node
    d = start_node.gene.shape[0]**2 # number of repetitions pr temperature value
    temperature = initial_temp( node )

    repetitions = 0
    while temperature>1e-100 and node.cost > goal_cost:
        neighbor = Node( node.random_neighbor(),k ) # choose random neigh
        if neighbor.cost < node.cost:
            node = neighbor
        else:
            annealing = random.uniform(0,1)
            if annealing < math.pow(math.e,-(neighbor.cost-node.cost)/temperature):
                """
                A stochastic evaluation for checking it its time to perform the
                annealing.
                """
                node = neighbor
        repetitions += 1

        if repetitions<d:
            # set a dependant initial temperature
            # we enter this clause during the first couple of runs
            temperature = dependant_initial_t( temperature,node,neighbor )
        elif repetitions%d==0:
            # and then we enter this clause occationally afterwards.
            temperature = reduce_t( temperature,repetitions )
    #end while
    return node
```

Figure 3: How this script handles the temperature variations.

2 The Egg Carton Puzzle

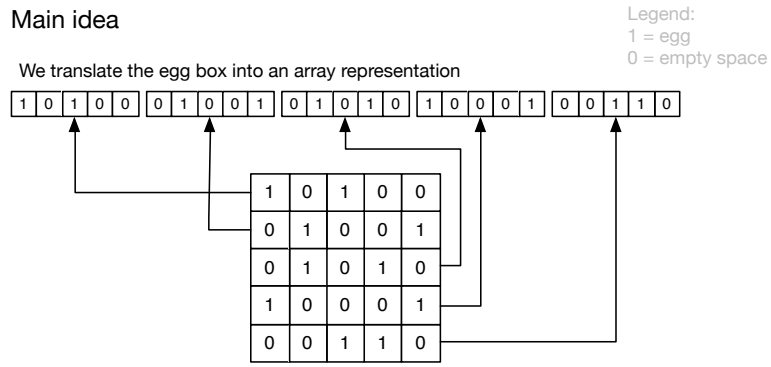
The specialized implementation

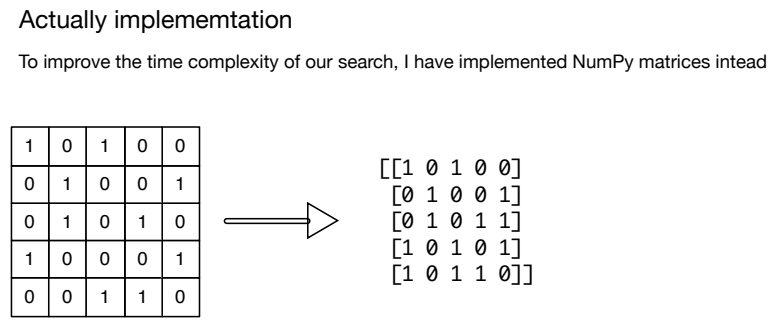
I made a few alterations to the generalized node to be able to constraining the k-value. I am also using the NumPy function shuffle to generate a new random child node. *Shuffle()* will move around the elements in the given row or column, and thus perform a "minor" alteration to the parent gene (as required).The idea behind the *objective function()* will be explained later.
My code required to further customizing to solve the assignment.

```
class Node:
    def __init__(self, gene, k):
        # The node must contain a cost value
        self.gene = gene
        self.k = k
        self.cost = self.objective_function( gene )

    def random_neighbour(self):
        # generate neighbors
        tmp = np.copy( self.gene )
        np.random.shuffle(tmp[random.random()*tmp.shape[0],:])
        np.random.shuffle(tmp[:,random.random()*tmp.shape[0]])
        return tmp

    def objective_function(self, gene ):
        n = gene.shape[0]
        k = self.k
        horizonal = np.sum(gene, axis=1)-k
        vertical = np.sum(gene, axis=0)-k
        diag1 = np.array([
                            sum(gene.diagonal(k-n+i))
                            for i in xrange(2*(n-k)+1)
                        ])-k
        diag2 = np.array([
                            sum(np.fliplr(gene).diagonal(k-n+i))
                            for i in xrange(2*(n-k)+1)
                        ])-k

        return  np.sum(horizonal[horizonal>0],axis=0) + \
                np.sum(vertical[vertical>0],axis=0) + \
                np.sum(diag2[diag2>0],axis=0) + \
                np.sum(diag1[diag1>0],axis=0)
```

Figure 4: The python implementation of the specialized search node

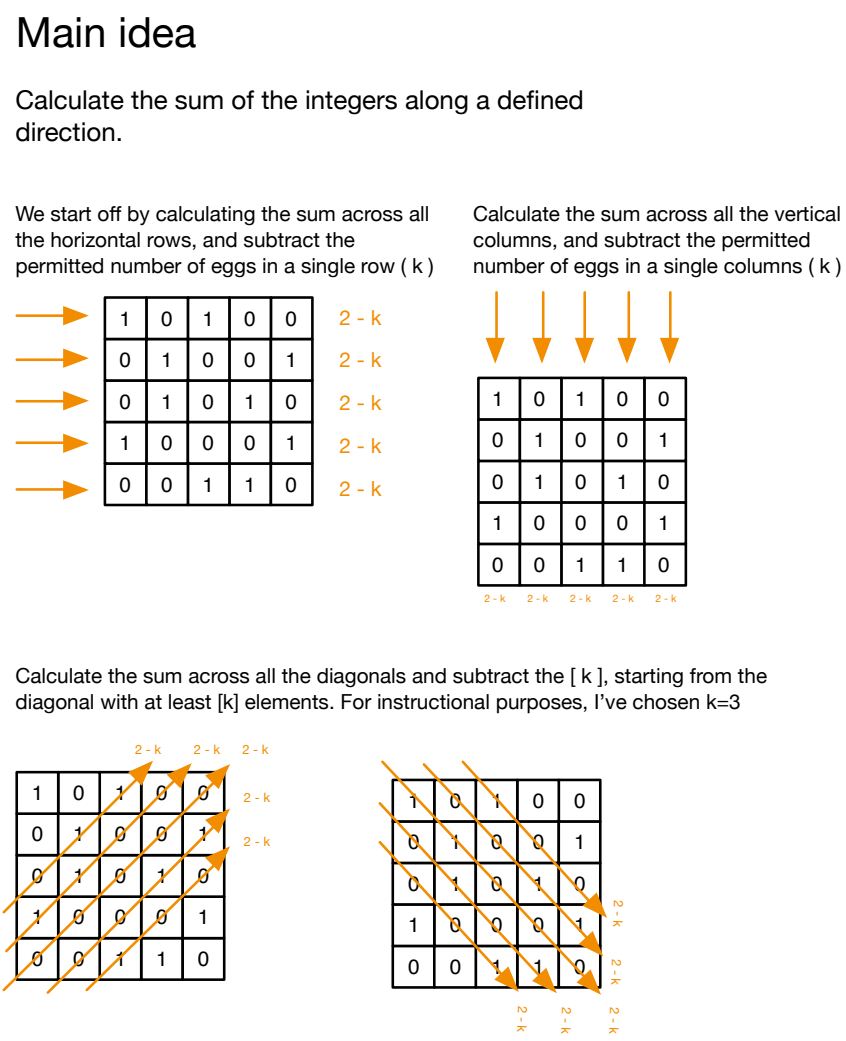
The problem representation





I chose this implementation strategy because it would make it a lot easier for me to perform the evaluation in the objective function. When represented like this, all we need to do is sum up the integers along a defined line.

The objective function



This function will always return a positive value, and only return 0 if we have an optimal solution. It is computationally fast, but horrific in memory if we were to encouter BIG boxes.

The execution results

```
Simulated Annealing - (5x5) box of 10 eggs with k=2:
[[0 1 0 1 0]
 [0 0 0 1 1]
 [0 1 0 0 1]
 [1 0 1 0 0]
 [1 0 1 0 0]]

Simulated Annealing - (6x6) box of 12 eggs with k=2:
[[0 0 1 1 0 0]
 [1 0 0 0 1 0]
 [0 0 1 0 1 0]
 [1 1 0 0 0 0]
 [0 0 0 1 0 1]
 [0 1 0 0 0 1]]

Simulated Annealing - (8x8) box of 8 eggs with k=1:
[[0 0 1 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 0 1 0 0]]

Simulated Annealing - (10x10) box of 29 eggs with k=1:
[[1 0 0 0 1 0 0 1 0 0]
 [0 0 0 1 0 1 0 1 0 0]
 [0 0 1 1 0 1 0 0 0 0]
 [1 0 1 0 0 0 0 0 1 0]
 [0 0 0 0 0 1 1 0 0 1]
 [0 0 0 0 0 0 1 0 1 0]
 [1 0 0 0 1 0 0 0 0 1]
 [0 0 1 0 0 0 1 0 0 1]
 [0 1 0 0 1 0 0 0 1 0]
 [0 1 0 1 0 0 0 1 0 0]]
```

Discuss the similarities and differences between heuristics and objective functions.

A heuristic function serves as a means to describing how far this state is from the goal state. It has a definition of a goal to reach whilst a objective function looks at the given state objectively, and tells us how good this state is. The term "objective function" is used when it returns a value we want to maximize or minimize (as is done in this assignment).

The Python script

```
import random
import math
import itertools
import numpy as np

"""
minimizing

Observation:
* each node should generate the same number of neighbors
* be aware of poor performance under spiky neighborhoods
* yields better results than local search and multisstart search after t time
* should not be used on TSP or similar problems

This example will sort the digits 0123456789 #note that 0 will be removed from the integer
"""

class Node:
    def __init__(self, gene, k):
        # The node must contain a cost value
        self.gene = gene
        self.k = k
        self.cost = self.objective_function( gene )

    def random_neighbour(self):
        # generate neighbors
        tmp = np.copy( self.gene )
        np.random.shuffle(tmp[random.random()*tmp.shape[0],:])
        np.random.shuffle(tmp[:,random.random()*tmp.shape[0]])
        return tmp

    def objective_function(self, gene ):
        n = gene.shape[0]
        k = self.k
        horizonal = np.sum(gene, axis=1)-k
        vertical = np.sum(gene, axis=0)-k
        diag1 = np.array([
                            sum(gene.diagonal(k-n+i))
                            for i in xrange(2*(n-k)+1)
                        ])-k
        diag2 = np.array([
                            sum(np.fliplr(gene).diagonal(k-n+i))
                            for i in xrange(2*(n-k)+1)
                        ])-k

        return  np.sum(horizonal[horizonal>0],axis=0) + \
                np.sum(vertical[vertical>0],axis=0) + \
                np.sum(diag2[diag2>0],axis=0) + \
                np.sum(diag1[diag1>0],axis=0)

def initial_temp( node ):
    tmp = np.ones(shape=node.gene.shape,dtype="uint8")
    return float(node.objective_function( tmp ))

def dependant_initial_t( temperature,node,neighbor ):
    # Increase the temperature to make this neighbor a very likely choice
    # This function may not decrease the temperature

    x = temperature/(abs(node.cost-neighbor.cost)+1)*math.log1p(100/99)
    return temperature/x if 1>x>0 else temperature

def reduce_t( temperature,repetitions ):
    # common function for temperatur reduction
    return temperature/math.log(repetitions+2,2)

def sa( start_node, goal_cost, k ):
    node = start_node
    d = start_node.gene.shape[0]**2 # number of repetitions pr temperature value
    temperature = initial_temp( node )

    repetitions = 0
    while temperature>1e-100 and node.cost > goal_cost:
```

```
        neighbor = Node( node.random_neighbour(),k ) # choose random neigh
        if neighbor.cost < node.cost:
            node = neighbor
        else:
            annealing = random.uniform(0,1)
            if annealing < math.pow(math.e,-(neighbor.cost-node.cost)/temperature):
                node = neighbor
        repetitions += 1

        if repetitions<d:
            # set a dependant initial temperature
            temperature = dependant_initial_t( temperature,node,neighbor )
        elif repetitions%d==0:
            temperature = reduce_t( temperature,repetitions )
    #end while
    return node

n=6 # box size
x = 12 # eggs
k = 2
gene = np.array([1]*x+[0]*(int(n**2)-x)).reshape(n,n)

print "Simulated Annealing - (%dx%d) box of %d eggs width k=%d:" % (n,n,x,k)

print sa(
            Node( gene,k ),
            goal_cost = 0,
            k = k
        ).gene
```