CS5330 - Randomized Algorithm
Project Report

Due on Thursday, October 16, 2014

Dr. Lee Hwee Kuan - Thu, 06:30pm

LIU Weizhi *

Department of Industrial & Systems Engineering
National University of Singapore

November 13, 2014

*weizhiliu2009@gmail.com

Maximizing Network Reward Based on A Genearal Framework of Monte Carlo Tree Search

Abstract

This study implemented the general framework of MCTS to solve the network population problem. Preliminary comparision of different algorithms demonstrates that uct0.5 and uct1.0 perform best in both criteria of best rewards detection and computation time, especially, uct seems like possessing the abality of learning. While rmc and nmc1 performs much faster than the other algorithms, their abality of detecting best population sequence is poor. An interesting finding is that there might exist some periods for nmc2 algorithms in terms of reward sequence.

Keywords: Monte Carlo Tree Search, Multi-armed Bandit Problem, Network Population

1 Introduction

The goal of the project is to search the optimal network population sequence in order to maximize the total reward of the network given two input files, namely an undirected network adjacent list and original color sequence. The network adjacent list file depicts which node connects to the other node, and the network may contain self-loop and multiple edges. The original color sequence is a 0/1 binary sequence, which illustrates the order of the color sequence (eg. 0 represents red, 1 represents blue). The total reward of the network is calculated by summing up all the numbers of edges which connect to two different color nodes (note that we only consider colored node, and the uncolored nodes have no color at all). At the initial stage of the game, one should pick an arbitrary node from the network and color it according to the corresponding color from the original color sequence. Then the next population candidate could only be selected from those nodes which are not colored yet and connect to at least one already colored node. The game will enter into termial state when the original color sequence runs out or there are no further possible candidate nodes to color. All in all, the aim of this game is to select an optimal enough population sequence to maximize the total reward of the network.

Obviously, when the graph is large enough, the naive brute force algorithm could consume enormous time which is not acceptable given a limited computation budget. One possible and efficient way to solve this game is to implement Monte Carlo Tree Search (MCTS) algorithms which have gained remarkable attentations in the past few years, especially after the significant success on the game of Go [1]. However, there are various MCTS algorithms (eg. UCT [2], Nested Monte Carlo (NMC) search [3], Reflexive Monte Carlo (RMC) search [4]) for which may only perform well on some certain problems. Therefore, for a specificed problem, a domain based algorithm will be designed to best fit that problem. However, it's very difficult to design a domain based algorithm which needs more deep understanding of the original problem. To facilitate this process, this paper implemented a more general framework of MCTS which can generate all possible popular MCTS algorithms nowadays based on the work of Francis Maes et al. ([5]). The remainder of this paper are as follows. The general framework of MCTS is proposed in section 2, followed by the comparision results and current network best rewards for different sets in section 3. Finally, we conclude this paper by summarizing the important facts and possible future work.

2 Method

The important notations of this paper is listed in Table 1. In this section, the network construction and reward evaluation is illustrated firstly and then the general framework of MCIS will be discussed.

Table 1: Notations

notation	definition
$\mathcal{A}_{n\times n}$	adjacent matrix of the network with n nodes
a_{ij}	element of $\mathcal{A}_{n\times n}$ which represents the number of edges between node i and j
$\vec{s}$	the original input sequence
$\vec{c}_{1\times n}$	color vector for every nodes, c_i represents the color of node i
r_{ij}	reward value between node i and node j
$R(\mathcal{A}_{n\times n}, \vec{c}_{1\times n})$	total reward of the network
R^*	current best network reward
w_k	the candidate nodes set at step k
$\vec{p}_k = (p_1, p_2, \cdots, p_k)$	population sequence at step k of which element p_i represents the ith populated nodes
$\vec{p}^*$	current best population sequence
$\tau(k)$	represents if the game enters into end at time k
B	total budget for each algorithm
$numCalls$	current times of evaluation
$\mathcal{S}$	search component
$N^{(repeat)}$	repeat times for repeat component
$N^{(select)}$	multiple factor for select component
η	weight factor for exploring of ucb value
$\mathcal{L}_i$	the lower level search component standalone parameters recursive list at level i

2.1 Network Construction and Reward Evaluation

The network $\mathcal{A}_{n\times n}$ could be constructed based on the network adjacent list by continuously update a_{ij}. Since the network is undirected, a_{ij} should be the same with a_{ji}.

The ith elemet of color vector $\vec{c}_{1\times n}$ equals to the following value conditioning on the color of the ith node

$$c_i = \begin{cases} -1 & \text{if node } i\text{'s color is 0} \\ 0 & \text{if node } i \text{ is not colored} \\ 1 & \text{if node } i\text{'s color is 1} \end{cases} \tag{1}$$

Thus, we can easily calculate the reward between node i and node j given a network $A_{n\times n}$ and color vector $\vec{c}_{1\times n}$ based on the equation (2)

$$r_{ij} = \frac{(c_i c_j - 1)}{2} c_i c_j a_{ij} = \frac{1}{2}(c_i^2 a_{ij} c_j^2 - c_i a_{ij} c_j) \tag{2}$$

Consequently, the total reward of the network could be formulated in a matrix expression (also note that the network $\mathcal{A}_{n\times n}$ is symmetric)

$$R(\mathcal{A}_{n\times n}, \vec{c}_{1\times n}) = \frac{1}{4}[(\vec{c}_{1\times n})^2 \mathcal{A}_{n\times n} (\vec{c}^T_{1\times n})^2 - \vec{c}_{1\times n} \mathcal{A}_{n\times n} \vec{c}^T_{1\times n}] \tag{3}$$

2.2 General Framework of MCTS

The general framework of MCTS in this article consists of five helper components and five search components. The detailed description of each component is presented at the following parts. In addition, an algorithm generator should be designed which could create and implement many MCTS algorithms after specifying the recursive relationships between search componets and necessary parameters.

2.2.1 Helper Components

Table 2 depicts the five helper componets and their corresponding task. A more detailed implementation of this componets could be seen in the Appendix B.

Table 2: Illustration of helper components

helper component	task	input	output
candidate	update candidate nodes set for populating by set operation rather than loop	$\vec{p}_{k-1}, p_k, w_{k-1}$	w_k
terminal	check wheter the game enters into end	$\vec{p}_k, w_k$	$\tau(k)$
reward	calculate the network reward	$\mathcal{A}_{n\times n}, \vec{c}_{1\times n}$	R
evaluate	update the budget consumption, best reward and population sequence	$\vec{p}_k, \vec{p}^*, R^*, \mathcal{A}_{n\times n}, \vec{c}_{1\times n}$ $numCalls, B$	$\vec{p}^*, R^*, numCalls$
invoke	invoke other search components	$\vec{p}_k, w_k, \mathcal{S}$	$\vec{p}_m(m > k)$

2.2.2 Search Components

Table 3 illustrates the five search componets and their corresponding task. Note that the input of all five search components include the current population sequence $\vec{p}_k$ and next candidate node set w_k and the output should be a best full or partial population sequence if the search components generate and evaluate many possible population sequences, or just a population which might not be best (eg. simulate component just return a uniformly randomly selected full population sequence). In addition, the repeat component should receive another parameter, namely the total repeat times $N^{(repeat)}$, while select component contains two more input parameters, namely the multiple factor $N^{(select)}$ and the weight factor η of exploring for the ucb value. Furthermore, in order to generate more algorithms recurisively, the study has proposed two different types of search components with regard to whether it can invoke the other search component. One is **atom component** which can not invoke the other search component (eg. simulate component), and the other one is **free component** (in this framework, eg. step, repeat, lookahead, select) which can invoke the other component even themselves. Those free component, compared with the atom component, include another input parameter called **lower level search component standalone parameters recurisive list** which contains all the standalone parameters and further **lowe level search component standalone parameters recursive list** for the lower level search component. The lowe level search component represents the the set of all further invoking search components of the current search component. For example, if one algorithm is like step(repeat(select(simulate(), N^{select}, η), N^{repeat})), then the lower level search component of repeat component is just select and simulate.

Most of search components here are same with those of Maes et al. ([5]) despite the select component. There are basically two major differences

- the budget for select component is automatically adjusted according to the size of graph and sequence, and the step of current population sequence which can be formulated as below at step k

$$Budget(k)^{select} = Size(w_k)[(1 - \frac{Dim(\mathcal{A}_{n\times n})N^{select}}{Size(w_k)})\frac{Size(\vec{p}_k)}{Size(\vec{s})} + \frac{Dim(\mathcal{A}_{n\times n})N^{select}}{Size(w_k)}] \tag{4}$$

  Actually, the initial budget is just $Dim(\mathcal{A}_{n\times n})N^{select}$ and the last budget is only $Size(w_k)$. Therefore, this automatic budget allocation will efficiently reduce the budget resoruces when the algorithms have explored many possible population sequences.

- Since the reward part and explore part of ucb value in this problem is extremely different with regard to the scale, therefore it's necessary to normalize the two parts. Reward part is divided by the current best rewatd, while the explore part is transformed into $(0, 1)$ via logistic function. However, since the possible population sequence space is really large, and it might be impossible to find the best population sequence which means normalize the two parts will increase the possibliy of exploring which is not efficient. After finding that new ucb value doesn't improve the best reward, so I finally give up this approach and implement the traditional way (actually, I think the original reward part of ucb value should be scaled into $[0, 1]$).

Table 3: Illustration of search components

search component	task	type
simulate	uniformly randomly select a full population sequence	atom component
step	generate a full population sequence step by step	free component
repeat	return the best population by repeating N^{repeat} times evaluation	free component
lookahead	return the best population by evaluating full population sequence among all next candidates	free component
select	a mini version of UCB	free component

The detailed implementation of each search component can be seen in the Appendix B.

2.2.3 Algorithms Generator

Based on the help of basic search component, dozens of MCTS algorithms could be defined. For example, the possible popular MCTS algorithms are listed in Table 4. Many other possible algorithms could be seen

Table 4: MCTS algorithms examples based on the general framework

algorithm	recursive expression
rmc($N_1^{select}, N_2^{select}$)	step(repeat(N_1^{select}, step(repeat(N_2^{select}, simulate()))))
nmc(1)	step(lookahead(simulate()))
nmc(2)	step(lookahead(step(lookahead(simulate()))))
uct($N^{repeat}, N^{select}, \eta$)	step(repeat(N^{repeat}, select(N^{select}, η, simulate())))

in Maes et al.[5]. Actually, the recursive expression could be implemented by **functional programming**. However, I have little knowledge about that, but I have figured out my way to implement the recursive expression by the **lower level search component standalone parameters recursive list** mentioned before. The implementation of the algorithms generator based on python is revealed in the next section.

3 Results & Discussion

3.1 Datasets

There are 7 datasets and the size of network and original sequence (which will influence the population search space) are illustrated in Table 5. Consequently, a possible simulation path should be set 1, set 2, set 3, set 6, set 4, set 5 and set 7 considering the difficulty of different datasets.

3.2 Algorithms Comparisions

To decide which algorithms (one or more) may be more efficient to solve this problem, five basic algorithms were evaluated in the terms of optimality and computation time based on set 2. The five algorithms include

Table 5: Descriptions of datasets

dataset	network size (range)	sequence size
set 1	10	10
set 2	153	20
set 3	153	130
set 4	961	400
set 5	5002	4000
set 6	483	400
set 7	11748	9000

rmc(10000,10), nmc1, nmc2, uct0.5 (which means the exploring weight factor is 0.5), uct1.0. The comparision of optimality could be seen in Figure 1, while Figure 2 depicits the difference on computation time. For each algorithm, 10000 budget are allocated to it and 10 independently simulation runs are conducted. Apart from this, the trend of best reward against each full population evaluation is illustrated in Figure 3 - 7. Though nmc1 is the most fast algorithms, its abality to find the optimal population sequence is not so good. While uct0.5 and uct1.0 performs best in both criterias of best rewards and computation time. In addition, note that uct0.5 and uct1.0 seems to improve their performance continuously based on Figure 6 and Figure 7. Consequently, this study implement these two algorithms and then find the best population sequence from them. Another remarkable finding is that the there might exists some periods in the reward sequence of nmc2 according to Figure 5.

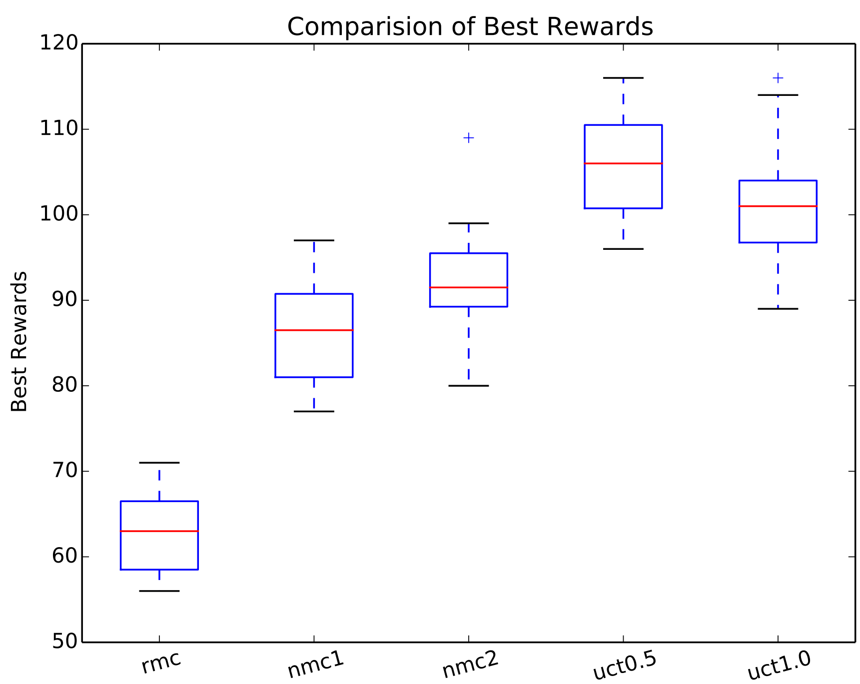

Figure 1: Comparision of Best Rewards

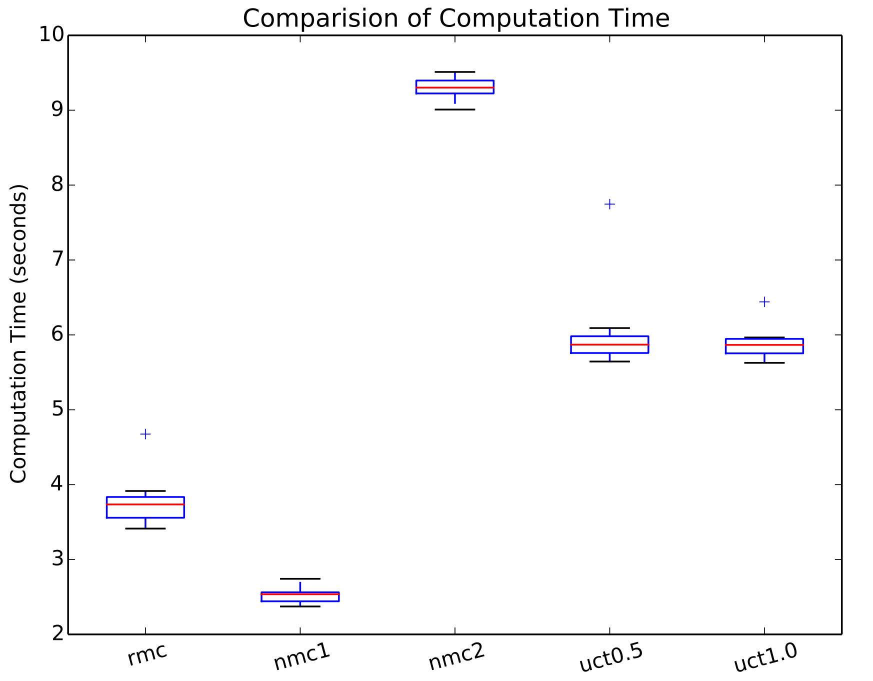

Figure 2: Comparision of Computation Time

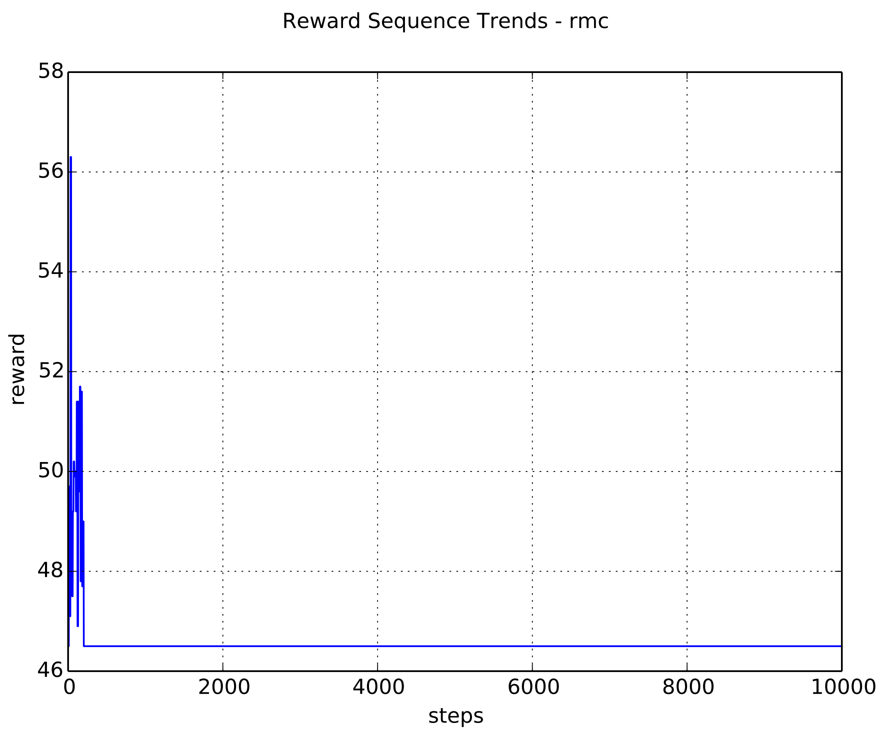

Figure 3: Trends of Reward Sequence for rmc

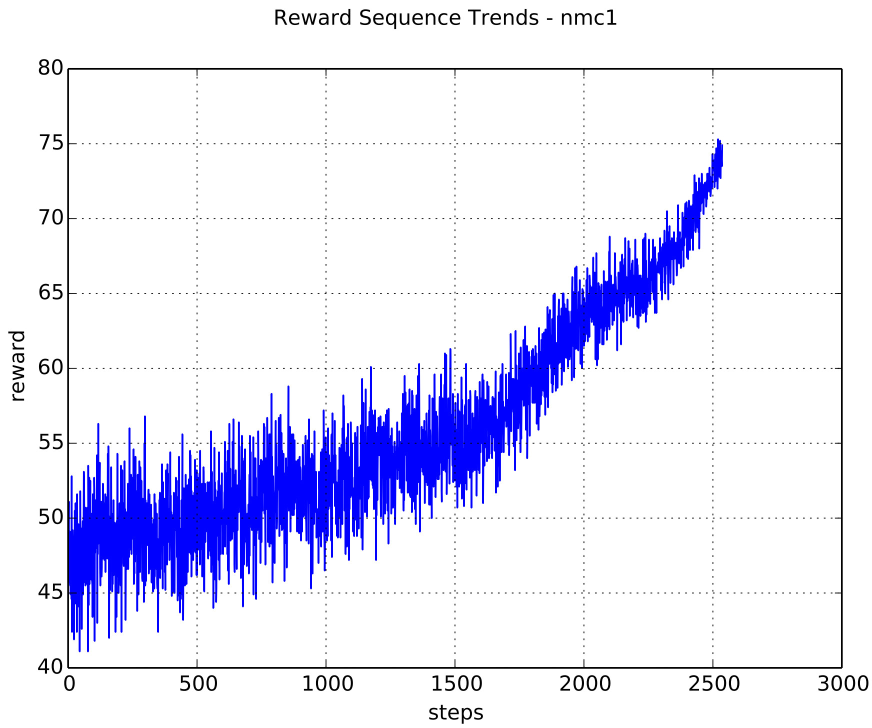

Figure 4: Trends of Reward Sequence for nmc1

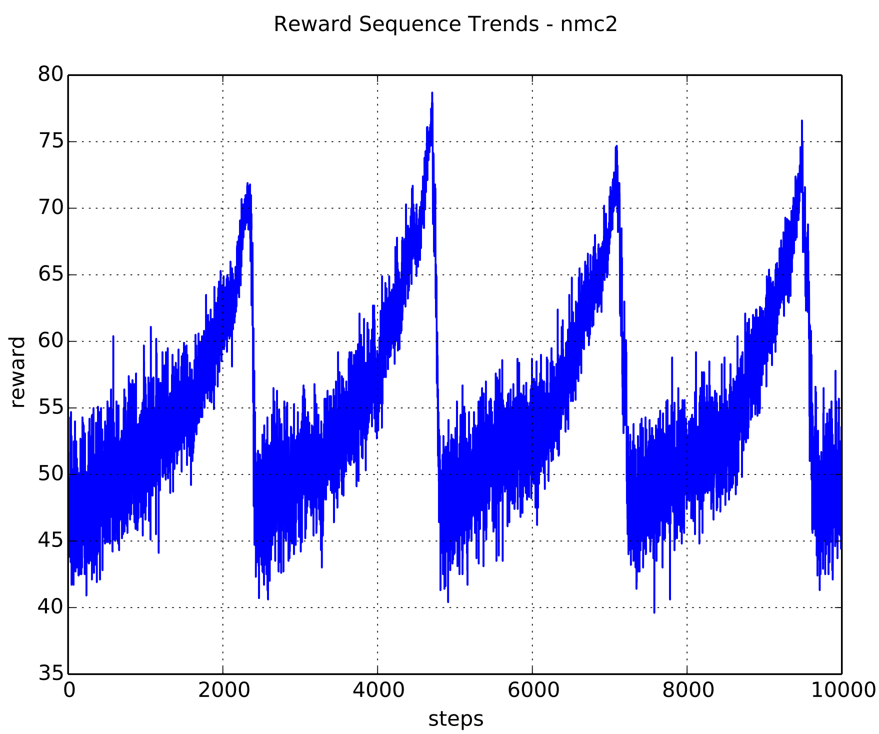

Figure 5: Trends of Reward Sequence for nmc2

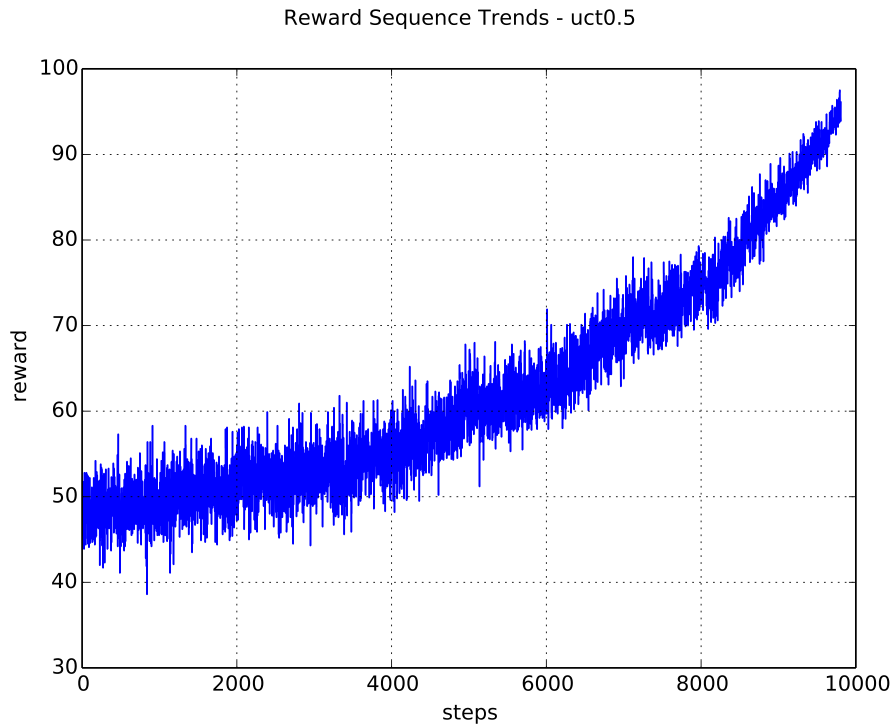

Figure 6: Trends of Reward Sequence for uct0.5

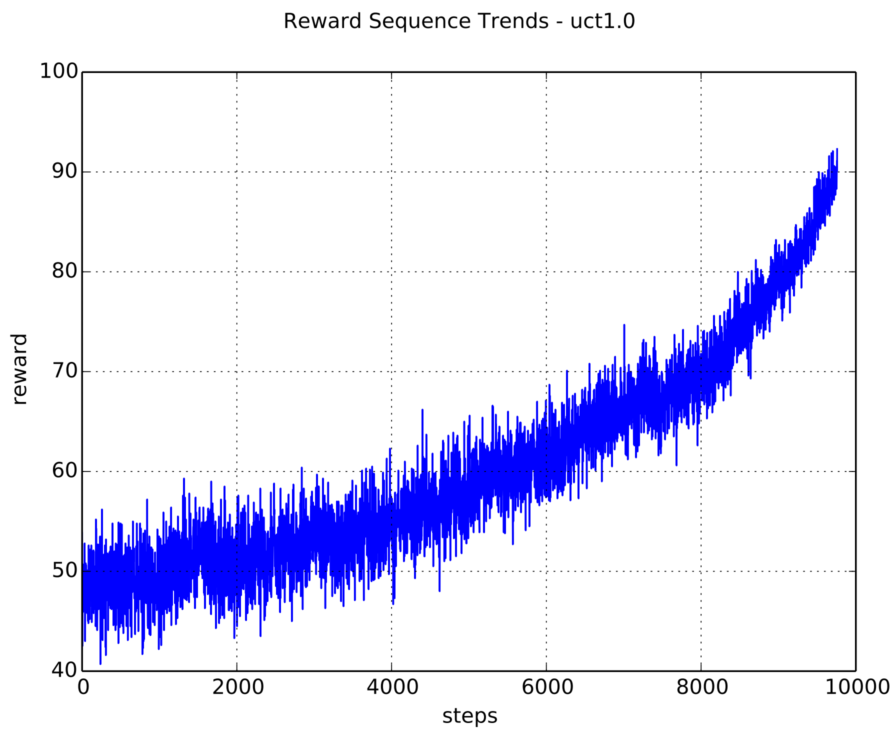

Figure 7: Trends of Reward Sequence for uct1.0

3.3 Best Reward

The current best rewards for each datasets are in Table 6.

Table 6: Illustration of search components

dataset	current best reward
set 1	19
set 2	157
set 3	2612
set 4	248
set 5	1170
set 6	587
set 7	9979

4 Conclusion

This study implemented the general framework of MCTS to solve the network population problem. Preliminary comparision of different algorithms demonstrates that uct0.5 and uct1.0 perform best in both criterias of best rewards detection and computation time, especially, uct seems like possessing the abality of learning. Future work may focus on the more detailed comparisions between different algorithms (eg. based on a meta multi-armed bandit problems). Also, a fast and efficient approach to detect new candidate nodes set and calculate the total reward of network should be explored. Most importantly, rather than treating select search component as a simple multi-armed bandit problem, more infomation should also be recored for the deeper nodes. Lastly, the better understanding of the network and original color sequence is indispensable. Due to time limitation, this work still needs more effort to improve the best reward and detect more efficient algorithm based on the general framework of MCTS.

References

[1] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2007.

[2] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[3] Tristan Cazenave. Nested monte-carlo search. In *IJCAI*, volume 9, pages 456–461, 2009.

[4] Tristan Cazenave. Reflexive monte-carlo search. 2007.

[5] Francis Maes, David Lupien St-Pierre, and Damien Ernst. Monte carlo search algorithm discovery for one player games. *arXiv preprint arXiv:1208.4692*, 2012.

Appendix A - Comparision of Algorithms

Table 7: Comparision of Best Rewards

algorithm	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10	mean	std
rmc	71	63	71	63	56	58	67	60	58	65	63.20	5.06
nmc1	81	93	97	80	90	77	91	90	83	81	86.30	6.34
nmc2	99	94	89	96	80	84	92	109	91	90	92.30	7.61
uct0.5	109	114	111	100	100	103	116	103	109	96	106.10	6.30
uct1.0	104	89	114	100	116	102	96	99	94	104	101.80	7.93

Table 8: Comparision of Computation Time (seconds)

algorithm	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10	mean	std
rmc	4.675	3.800	3.553	3.413	3.568	3.671	3.840	3.820	3.915	3.354	3.78	0.33
nmc1	2.429	2.541	2.550	2.566	2.531	2.373	2.441	2.742	2.445	2.656	2.53	0.11
nmc2	9.260	9.407	9.511	9.202	9.009	9.344	9.255	9.366	9.435	9.214	9.30	0.14
uct0.5	5.746	5.793	7.746	5.644	5.916	6.091	5.679	5.927	5.823	6.000	6.04	0.59
uct1.0	5.873	5.833	5.952	5.626	5.929	5.662	5.859	6.441	5.727	5.965	5.89	0.22

Appendix B - Source Code

Code Listing 1: A General Framework of MCTS based on Python

```
# -*- coding: utf-8 -*-
"""
Created on Wed Oct  8 12:45:20 2014

@author: liuweizhi
"""
import os,sys,glob
import random
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import math
import copy
import shelve
import easygui

class GameTree():
    def __init__(self):
        self.graph = []
        self.sequence = []
        return
    def initialization(self, graphdir, seqdir):
        ''' return the constructed graph and sequence '''
        ## build the network
        f_graph = open(graphdir, 'r')
        content = f_graph.read().replace('\n', ' ').strip(' ').split(' ')
        content = [int(foo) for foo in content]
```

```
        scale =  max(content) - min(content) + 1
        f_graph.close()
        ### initialize the adjacent network with scale * scale (graph[i,j] means the
        ### number of edges between node i and node j)
        graph = np.matrix([[ 0 for j in range(scale)] for i in range(scale)])
        f_graph = open(graphdir, 'r')
        for line in f_graph:
            tmp = line.strip('\n').split(' ')
            try:
                [v1, v2] = [int(tmp[i]) - min(content) for i in range(len(tmp))]
                if (v1 != v2):
                    graph[v1,v2] = graph[v1,v2] + 1
                    graph[v2,v1] = graph[v2,v1] + 1
                else:
                    graph[v1,v2] = graph[v1,v2] + 1
            except:
                print "the sequence file isn't complete"
                sys.exit(0)
        self.graph = graph
        f_graph.close()
        ## record the sequence
        sequence = []
        f_seq = open(seqdir, 'r')
        content = f_seq.read().strip('\n')
        for color in content:
            color = int(color)
            if color == 0:
                sequence.append(-1)
            elif color == 1:
                sequence.append(1)
            else:
                print "the color sequence contains number other than 0 or 1!!!!"
                sys.exit(0)
        self.sequence = sequence
        f_seq.close()
        return [self.graph, self.sequence]

class MCTS():
    def __init__(self):
        self.name = 'Noname'
        self.best_population = []
        self.best_reward = -9999
        self.graph = []
        self.sequence = []
        self.setdir = ''
        return

    def initialization(self, graph, sequence, setdir):
        self.best_population = []
        self.best_reward = -9999
        self.graph = graph
        self.sequence = sequence
        # self.setdir is for the self.output() method
        self.setdir = setdir
        return

    def step(self, population, candi, func_list):
        ''' given unifinished population, return the best population and best reward
        which are constructed step by step '''
```

```
        # generate the population step by step
        id = len(population) + 1
        while not(self.terminal(population, candi)):
            population = self.invoke(population, candi, func_list)
            # update the next candidate space
            candi = self.candidate(population[0:id-1], [population[id-1]], candi)
            # only obtain the id th element from the population derived from
            # self.invoke, step by step
            population = population[0:id]
            id = id + 1
        # update the best_population and best_reward
        self.evaluate(population)
        return population

    def repeat(self, population, candi, func_list, N):
        ''' repeat n simulations using given search component and return the best
        population sequence '''
        best_population = []
        best_reward = -9999
        for i in range(N):
            tmp_population = self.invoke(population, candi, func_list)
            tmp_reward = self.reward(tmp_population)
            if tmp_reward > best_reward:
                best_population = tmp_population
                best_reward = tmp_reward
        return best_population

    def lookahead(self, population, candi, func_list):
        ''' return the all possible population for the next step '''
        best_population = []
        best_reward = -9999
        for new_node in candi:
            # generate the new_population by append the new node to original population
            new_candi = self.candidate(population, [new_node], candi)
            new_population = [foo for foo in population]
            # just lookahead
            new_population.append(new_node)
            # further generate a new population sequence given new_population
            tmp_population = self.invoke(new_population, new_candi, func_list)
            tmp_reward = self.reward(tmp_population)
            # update the best_population and best_reward
            if tmp_reward > best_reward:
                best_population = tmp_population
                best_reward = tmp_reward
        return best_population

    def select(self, population, candi, N, thr, func_list):
        ''' given population, then select the next population state using UCB with
        explorer parameter thr and budget N * len(candi)'''
        best_population = []
        best_reward = -9999
        # initialize the reawrd and number of trials of the total len(candir) arms
        arms = [{'reward':0, 'trial':0} for i in range(len(candi))]
        ucb = [0 for i in range(len(arms))]
        total_trials = 0
        untried = [foo for foo in candi]
        # the total budget is N
        candi_budget = len(candi)
        population_budget = int((1 - (self.graph.shape[0] * float(N)) / len(candi)) *
```

```
              (len(population) / float(len(self.sequence))) +  (self.graph.shape[0] *
               float(N)) / len(candi))
        for i in range(candi_budget * population_budget):
            # exists some untried arms
            if untried:
                new_node = untried[int(random.uniform(0,len(untried)))]
                untried.remove(new_node)
            # all arms were tried, then using the ucb value to select next state
            else:
                new_node = candi[ucb.index(max(ucb))]
            # generate the next population
            sub_candi = self.candidate(population, [new_node], candi)
            sub_population = [foo for foo in population]
            sub_population.append(new_node)
            tmp_population = self.invoke(sub_population, sub_candi, func_list)
            tmp_reward = self.reward(tmp_population)
            # update best_population and best_reward
            if tmp_reward > best_reward:
                best_population = tmp_population
                best_reward = tmp_reward
            # update the attributes of arms and calculate the ucb value
            arm_id = candi.index(new_node)
            arms[arm_id]['reward'] += tmp_reward
            arms[arm_id]['trial'] += 1.0
            total_trials += 1.0
            #reward_ucb = arms[arm_id]['reward'] / (algo.best_reward * arms[arm_id]['trial'])
            #trial_ucb = 1.0 / (1.0 + math.exp(- math.sqrt(2 * math.log(total_trials)
                        # / arms[arm_id]['trial'])))
            reward_ucb = arms[arm_id]['reward'] / arms[arm_id]['trial']
            trial_ucb = math.sqrt(2 * math.log(total_trials) / arms[arm_id]['trial'])
            ucb[arm_id] = reward_ucb + thr * trial_ucb
        return best_population

    def simulate(self, population, candi):
        ''' return a randomly population sequence given current unfinished population
        sequence '''
        graph = self.graph
#        # states_seq is empty which means the initial state of simulation
#        if [] == population:
#            new_node = candi[int(random.uniform(0,len(candi)))]
#            population.append(new_node)
        # generate the uniformly randomly simulation population sequence
        while(not(self.terminal(population, candi))):
            new_node = candi[int(random.uniform(0, len(candi)))]
            candi = self.candidate(population, [new_node], candi)
            population.append(new_node)
        self.evaluate(population)
        return population

    def candidate(self, population, new_node, candi):
        ''' return the next candidate space given the population '''
        # return the ajcacent vector of new_node
        if (population==[]) and (new_node==[]) and (candi==[]):
            candi = list(np.where(self.graph.sum(axis=0).A1>0)[0])
        elif (population==[]) and (new_node) and (candi):
            new_node_neighbor = []
            for foo in new_node:
                new_node_neighbor.extend(np.where(self.graph[foo,].A1>0)[0].tolist())
            candi = list(set(new_node_neighbor) - set(new_node))
```

```
        else:
            new_node_neighbor = []
            for foo in new_node:
                new_node_neighbor.extend(np.where(self.graph[foo,].A1>0)[0].tolist())
            try:
                candi = list((set(candi) | set(new_node_neighbor)) - set(new_node)
                        - set(population))
            except:
                print 'candi',candi
                print 'new_node_neighbor',new_node_neighbor
                print 'new_node',new_node
                print 'population',population
                sys.exit(0)
        return candi

    def evaluate(self, population):
        ''' return the best_population, best_reward of the upper search component
        and the population evaluated, and check if the budget is run out '''
        global numCalls
        global budget
        # calculate the reward given finished population sequence
        reward = self.reward(population)
        # update the best_reward and best_population
        if reward > self.best_reward:
            self.best_reward = reward
            self.best_population = [population]
        elif reward == self.best_reward:
            self.best_population.append(population)
        # update the budget usage
        numCalls = numCalls + 1
        print '%s - (%d/%s) - best reward: %d/%d' % (self.name, numCalls, str(budget),
               self.best_reward, reward)
        if numCalls == budget:
            print 'the budget has been run out'
            # output the current best population and reward
            # self.output(self.best_population[0], self.best_reward)
            sys.exit(0) # need modify
        return reward

    def invoke(self, population, candi, func_list):
        if not(self.terminal(population, candi)):
            func = func_list[0]
            population = func['func'](population, candi, **func['argv'])
        else:
            self.evaluate(population)
        return population

    def reward(self, population):
        ''' return the reward of the graph given population sequence'''
        graph = self.graph
        sequence = self.sequence
        # for color_seq, if the value == 0, then it means the corresponding node isn't
        # painted
        color_seq = np.matrix([0 for v in range(self.graph.shape[0])])
        for v in population:
            color_seq[0, v] = sequence[population.index(v)]
        return int(np.square(color_seq) * graph * np.square(color_seq).T - color_seq *
graph * color_seq.T) / 4
```

```
    def terminal(self, population, candi):
        ''' return whether entering the terminal state '''
        if (population):
            if (candi and (len(population)<len(self.sequence))):
                flag = False
            else:
                flag = True
        # the initial state of population
        else:
            flag = False
        return flag


    def output(self, population, reward):
        ''' write the reward and population sequence into txt files '''
        # write reward into file
        checker = ['Lim Wei Quan', 'Lim Wei Zhong', 'Zhang Chen']
        f_reward = open(os.path.join(self.setdir, 'reward.txt'), 'w')
        f_reward.write('reward=%d\n' % reward)
        for foo in checker:
            f_reward.write('%s\n' % foo)
        f_reward.close()
        # write population sequence and corresponding color sequence into file
        sequence = self.sequence
        f_population = open(os.path.join(self.setdir, 'population.txt'), 'w')
        for i in range(len(population)):
            f_population.write('%d %d\n' % (population[i], (self.sequence[i]+1)/2))
        f_population.close()
        return


def recursive(func_list):
    ''' return the self-loop recursive func_list given func_list '''
    for i in range(len(func_list)-1):
        func_list[i]['argv']['func_list'] = func_list[i+1:len(func_list)]
    return func_list

def combinator(func_list):
    ''' run the recurisive func '''
    func_list = recursive(func_list)
    func = func_list[0]
    result = func['func'](**func['argv'])
    return result

def restore_shelve(filename):
    ''' restore the workspace '''
    f_shelf = shelve.open(filename)
    for key in f_shelf:
        globals()[key] = f_shelf[key]
    f_shelf.close()
    return
### parameters initialization
random.seed(5330)
### read file
homedir = os.getcwd()
datadir = os.path.join(homedir, 'data/submission')
datalist = os.listdir(datadir)
for foo in datalist:
    # initialize the game tree
```

```
    if 'set' in foo:
        setdir = os.path.join(datadir, foo)
    else:
        continue
    graphdir = glob.glob('%s/graph*.txt' % setdir)[0]
    seqdir = glob.glob('%s/seq*.txt' % setdir)[0]
    tree = GameTree()
    [graph, sequence] = tree.initialization(graphdir, seqdir)

    # create MCTS() class algo to let the algo_list legal
    algo = MCTS()
    algo.initialization(graph, sequence, setdir)
    population = []
    candi = algo.candidate([], [], [])
    # specify the algorithms we want to use like UCT, Nested Monte Carlo Tree
    algo_list = {
       'rmc': [{'func':algo.step, 'argv':{'population':population, 'candi':candi}},
               {'func':algo.repeat, 'argv':{'N':10000}}, {'func':algo.step, 'argv':{}},
               {'func':algo.repeat, 'argv':{'N':10}}, {'func':algo.simulate, 'argv':{}}],
       'nmc1': [{'func':algo.step, 'argv':{'population':population, 'candi':candi}},
                {'func':algo.lookahead, 'argv':{}}, {'func':algo.simulate, 'argv':{}}],
       'r-nmc1': [{'func':algo.repeat, 'argv':{'population':population, 'candi':candi, 'N':100}},
                  {'func':algo.step, 'argv':{}}, {'func':algo.lookahead, 'argv':{}},
                  {'func':algo.simulate, 'argv':{}}],
       'nmc2': [{'func':algo.step, 'argv':{'population':population, 'candi':candi}},
                {'func':algo.lookahead, 'argv':{}}, {'func':algo.step, 'argv':{}},
                {'func':algo.lookahead, 'argv':{}}, {'func':algo.simulate, 'argv':{}}],
        # uct algorithms computation time : len(seq) * N_repeat * (N_graph * len(graph)
        # + 1) * len(seq) / 2
       'uct0.5': [{'func':algo.step, 'argv':{'population':population, 'candi':candi}},
                  {'func':algo.repeat, 'argv':{'N':1}},
                  {'func':algo.select, 'argv':{'N': 2, 'thr':0.5}},
                  {'func':algo.simulate, 'argv':{}}],
       'uct1.0': [{'func':algo.step, 'argv':{'population':population, 'candi':candi}},
                  {'func':algo.repeat, 'argv':{'N':1}},
                  {'func':algo.select, 'argv':{'N': 3, 'thr':1.0}},
                  {'func':algo.simulate, 'argv':{}}],
       'r1': [{'func':algo.select, 'argv':{'population':population, 'candi':candi, 'N':100, 'thr':0.5
              {'func':algo.step, 'argv':{}}, {'func':algo.repeat, 'argv':{'N':100}},
              {'func':algo.select, 'argv':{'N':1000, 'thr':0.5}},
              {'func':algo.simulate, 'argv':{}}],
    }

    # the candidate algorithms to be compared
    algo_candi = ['uct0.5', 'rmc', 'nmc1']
    best_population = []
    best_reward = -9999
    algo_result = []
    for foo in algo_candi:
        # define the basic parameter
        global budget
        budget = float('inf')
        global numCalls
        numCalls = 0
        # initial the algo (clear the previous algorithm's best_population and
        # best_reward in algo)
        algo.initialization(graph, sequence, setdir)
        algo.name = foo
        # run the algorithms based on the foo
```

```
        combinator(algo_list[foo])
        reward = algo.reward(algo.best_population[0])
        # update the best population and best reward
        if reward > best_reward:
            best_population = [foo for foo in algo.best_population[0]]
            best_reward = reward
        # record the result of algo based on foo
        algo_result.append(copy.deepcopy(algo))
    # output the global best population, best reward given the comparision of all
    # algorithms in algo_candi
    algo.output(best_population, best_reward)
    # save the current workspace
    f_shelf = shelve.open(os.path.join(setdir, 'workspace.out'), 'n')
    for key in dir():
        try:
            f_shelf[key] = globals()[key]
        except:
            print 'ERROR shelving: {0}'.format(key)
    f_shelf.close()

    # notify the programmer that the simulation is done
    print '\a'*7
    #easygui.msgbox("Simulation is Done", title="Message")
```