In [7]:
```
from abc import ABC, abstractmethod
from collections import defaultdict
import math
import networkx as nx
import numpy as np
import os
import random
from matplotlib import pyplot as plt


class MCTS:
    "Monte Carlo tree searcher. First rollout the tree then choose a mov
e."

    def __init__(self, exploration_weight=1):
        self.Q = defaultdict(int)  # total reward of each node
        self.N = defaultdict(int)  # total visit count for each node
        self.children = dict()  # children of each node
        self.exploration_weight = exploration_weight

    def choose(self, node):
        "Choose the best successor of node. (Choose a move in the game)"
        if node.is_terminal():
            raise RuntimeError(f"choose called on terminal node {node}")

        if node not in self.children:
            return node.find_random_child()

        def score(n):
            if self.N[n] == 0:
                return float("-inf")  # avoid unseen moves
            return self.Q[n] / self.N[n]  # average reward

        return max(self.children[node], key=score)

    def do_rollout(self, node):
        "Make the tree one layer better. (Train for one iteration.)"
        path = self._select(node)
        leaf = path[-1]
        self._expand(leaf)
        reward = self._simulate(leaf)
        self._backpropagate(path, reward)

    def _select(self, node):
        "Find an unexplored descendent of `node`"
        path = []
        while True:
            path.append(node)
            if node not in self.children or not self.children[node]:
                # node is either unexplored or terminal
                return path
            unexplored = self.children[node] - self.children.keys()
            if unexplored:
                n = unexplored.pop()
                path.append(n)
                return path
```

```
            node = self._uct_select(node)  # descend a layer deeper

    def _expand(self, node):
        "Update the `children` dict with the children of `node`"
        if node in self.children:
            return  # already expanded
        self.children[node] = node.find_children()

    def _simulate(self, node):
        "Returns the reward for a random simulation (to completion) of `
node`"
        #invert_reward = True
        reward = 0
        while True:
            #print(node.to_pretty_string())
            if node.is_terminal():
                return reward
            node = node.find_random_child()
            reward = node.get_reward()

    def _backpropagate(self, path, reward):
        "Send the reward back up to the ancestors of the leaf"
        for node in reversed(path):
            self.N[node] += 1
            self.Q[node] += reward
            #reward = 1 - reward  # 1 for me is 0 for my enemy, and vice
versa

    def _uct_select(self, node):
        "Select a child of node, balancing exploration & exploitation"

        # All children of node should already be expanded:
        assert all(n in self.children for n in self.children[node])

        log_N_vertex = math.log(self.N[node])

        def uct(n):
            "Upper confidence bound for trees"
            return self.Q[n] / self.N[n] + self.exploration_weight * mat
h.sqrt(
                log_N_vertex / self.N[n]
            )

        return max(self.children[node], key=uct)
```

In [8]:
```
class Node(ABC):
    """
    A representation of a single board state.
    MCTS works by constructing a tree of these Nodes.
    Could be e.g. a chess or checkers board state.
    """

    @abstractmethod
    def find_children(self):
        "All possible successors of this board state"
        return set()

    @abstractmethod
    def find_random_child(self):
        "Random successor of this board state (for more efficient simula
tion)"
        return None

    @abstractmethod
    def is_terminal(self):
        "Returns True if the node has no children"
        return True

#     @abstractmethod
#     def reward(self):
#         "Assumes `self` is terminal node. 1=win, 0=loss, .5=tie, etc"
#         return 0

#     @abstractmethod
#     def __hash__(self):
#         "Nodes must be hashable"
#         return 123456789

#     @abstractmethod
#     def __eq__(node1, node2):
#         "Nodes must be comparable"
#         return True
```

In [9]:
```
#generate random graph of preset size
def generate_graph(vertices):
    return nx.generators.random_graphs.gnp_random_graph(vertices,
                np.random.uniform(0,0.5))
```

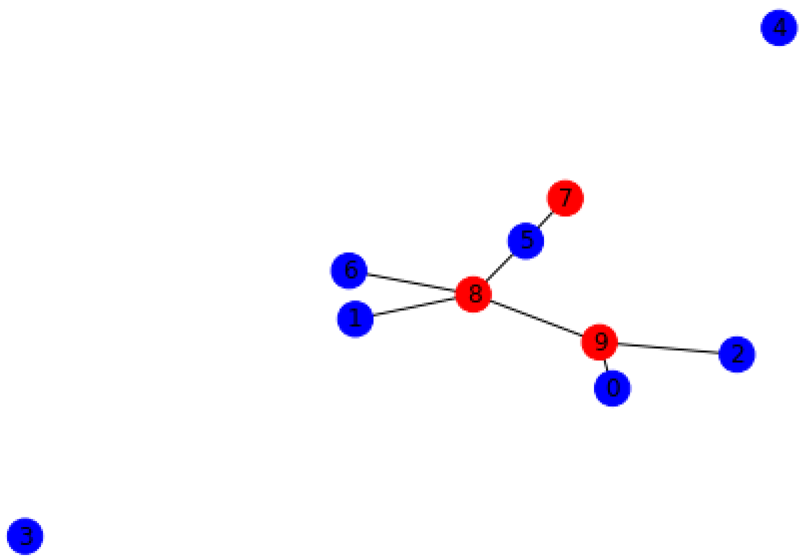
In [20]:
```
class VertexCoverInstance(Node):
    def __init__(self, graph, cover = [], reward = 0):
        self.graph = graph
        self.cover = cover
        self.reward = reward

    def find_children(self):
        possiblemoves = []
        if self.is_terminal():  # If the game is finished then no moves
 can be made
            return possiblemoves
        for i in list(self.graph.nodes):
            H = self.graph.copy()
            neigh = H.neighbors(i)
            step_reward = -1
            H.remove_node(i)
            H.add_node(i)
            possiblemoves.append(VertexCoverInstance(H, self.cover+[i],
self.reward+step_reward))
        return possiblemoves

    def find_random_child(self):
        if self.is_terminal():
            return None  # If the game is finished then no moves can be
 made
        temp = self.find_children()
        return random.sample(set(temp),1)[0]

#     def reward(board):
#         if not board.terminal:
#             raise RuntimeError(f"reward called on nonterminal board {b
oard}")
#         return self.reward #reward comes upon reaching terminal state

    def is_terminal(self):
        return nx.classes.function.is_empty(self.graph)

    def to_pretty_string(self):
        return str(list(self.graph.nodes()))

    def get_cover(self):
        return self.cover

    def get_reward(self):
        return self.reward
```

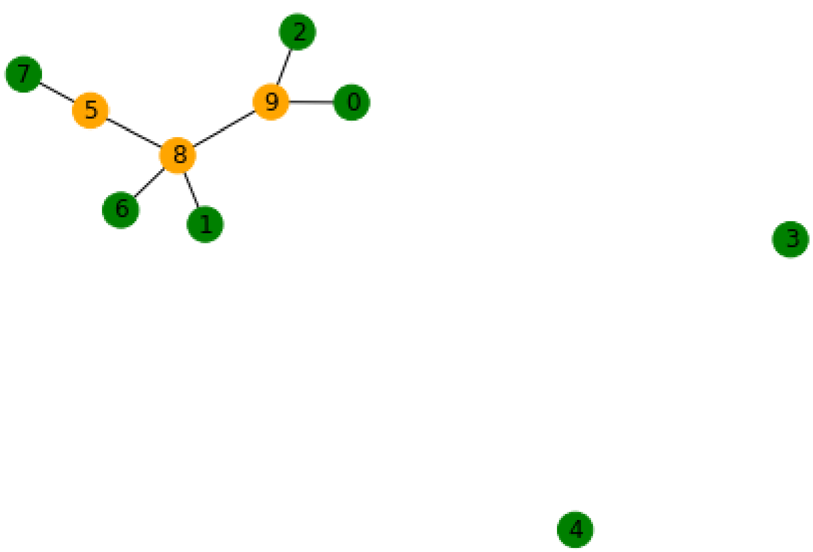
In [11]:
```
def play_game(G, display = True):
    tree = MCTS()
    board = VertexCoverInstance(G)
    #print(board.to_pretty_string())
    moves = 0
    while True:
        if board.is_terminal():
            break
        #80 rollouts per turn
        for _ in range(80):
            tree.do_rollout(board)
        board = tree.choose(board)
        #print(board.to_pretty_string())
        moves = moves+1
        if board.is_terminal():
            break

    vc = board.get_cover()
    if display:
        print("\n\nMCTS APPROX\n")
        print(vc)
        color_map = []
        for node in G:
            if node in vc:
                color_map.append('red')
            else:
                color_map.append('blue')
        plt.figure(1)
        nx.draw(G, node_color=color_map, with_labels=True)

    return vc
```

In [12]:
```
from networkx.algorithms import approximation as appx
import itertools

def show_optimal_vc(G, display=True):
    #opt = list(appx.vertex_cover.min_weighted_vertex_cover(G))
    def findsubsets(s):
        lists =[list(itertools.combinations(s, n)) for n in range(len(s
))]
        return list(itertools.chain.from_iterable(lists))

    powerset = findsubsets(list(G.nodes()))
    for s in powerset:
        H = G.copy()
        H.remove_nodes_from(s)
        if nx.classes.function.is_empty(H):
            opt = list(s)
            break
    if display:
        print("\n\nOPTIMAL\n")
        print(opt)
        color_map = []
        for node in G:
            if node in opt:
                color_map.append('orange')
            else:
                color_map.append('green')
        plt.figure(2)
        nx.draw(G, node_color=color_map, with_labels=True)

    return opt
```

In [13]:
```
G = generate_graph(10)
vc = play_game(G)
print()
opt = show_optimal_vc(G)
```

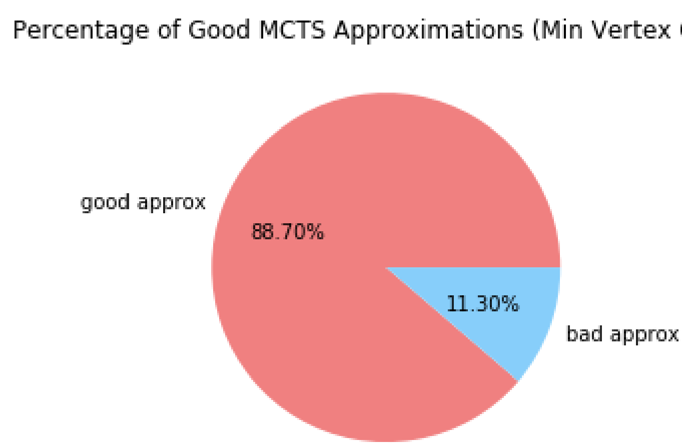
```
MCTS APPROX

[8, 7, 9]



OPTIMAL

[5, 8, 9]
```





In [14]:
```
NUM_SIMS = 1000

def sim_test_approx(simulations):
    count_success = 0
    for x in range(simulations):
        A = generate_graph(10)
        vc = play_game(A, display=False)
        opt = show_optimal_vc(A, display=False)
        if len(vc)<=len(opt)*1.1:
            count_success=count_success+1
    print(count_success)
    return count_success

count = sim_test_approx(NUM_SIMS)
plot_arr = [count,NUM_SIMS-count]
plt.figure(3)
plt.pie(plot_arr, colors = ['lightcoral','lightskyblue'], labels = ['goo
d approx', 'bad approx'],autopct='%1.2f%%')
plt.title('Percentage of Good MCTS Approximations (Min Vertex Cover)')
plt.show()
```

887



In [18]:
```
class MCTS_AZ:
    "Monte Carlo tree searcher with GNN. First rollout the tree then cho
ose a move."

    def __init__(self, net, exploration_weight=1):
        self.Q = defaultdict(int)  # total reward of each node
        self.N = defaultdict(int)  # total visit count for each node
        self.children = dict()  # children of each node
        self.priors = dict() # prior probability of visiting each child
 of a given node
        self.exploration_weight = exploration_weight
        self.net = net

    def choose(self, node):
        "Choose the best successor of node. (Choose a move in the game)"
        if node.is_terminal():
            raise RuntimeError(f"choose called on terminal node {node}")

        if node not in self.children:
            return node.find_random_child()

        def score(n):
            if self.N[n] == 0:
                return float("-inf")  # avoid unseen moves
            return self.Q[n] / self.N[n]  # average reward

        return max(self.children[node], key=score)

    def choose_by_policy(self, node):
        "Choose a successor of node according to policy"
        if node.is_terminal():
            raise RuntimeError(f"choose called on terminal node {node}")

        if node not in self.children:
            return node.find_random_child()

        policy = list(map(lambda n: self.N[n]/(self.N[node]-1), self.chi
ldren[node]))
        action = np.random.choice(len(policy), 1, p=policy)[0]
        successor = self.children[node][action]

        return successor,policy

    def do_rollout(self, node):
        "Make the tree one layer better. (Train for one iteration.)"
        path = self._select(node)
        leaf = path[-1]
        self._expand(leaf)
        reward = self._simulate(leaf)
        self._backpropagate(path, reward)

    def _select(self, node):
        "Find an unexplored descendent of `node`"
        path = []
        while True:
            path.append(node)
```

```
            if node not in self.children or not self.children[node]:
                # node is either unexplored or terminal
                return path
            for child in self.children[node]:
                if child not in self.children:
                    path.append(child)
                    return path
            node = self._uct_select(node)  # descend a layer deeper

    def _expand(self, node):
        "Update the `children` dict with the children of `node`"
        if node in self.children:
            return  # already expanded
        children = node.find_children()
        self.children[node] = children
        if children:
            self.priors[node] = self.net.predict(node)

    def _simulate(self, node):
        "Returns the reward for a random simulation (to completion) of `
node`"
        reward = 0
        while True:
            reward += node.get_reward()
            if node.is_terminal():
                return reward
            node = node.find_random_child()

    def _backpropagate(self, path, reward):
        "Send the reward back up to the ancestors of the leaf"
        for node in reversed(path):
            self.N[node] += 1
            self.Q[node] += reward

    def _uct_select(self, node):
        "Select a child of node, balancing exploration & exploitation"

        # All children of node should already be expanded:
        assert all(n in self.children for n in self.children[node])

        log_N_vertex = math.log(self.N[node])

        def uct(n):
            "Upper confidence bound for trees"
            i,n = n # expand from enumerate tuple
            return self.Q[n] / self.N[n] + self.exploration_weight * sel
f.priors[node][i] * math.sqrt(
                log_N_vertex / self.N[n]
            )

        return max(enumerate(self.children[node]), key=uct)[1]
```

In [2]:
```
def play_game_AZ(G, net,num_rollouts=80, display = True):
    tree = MCTS_AZ(net)
    board = VertexCoverInstance(G)
    while True:
        if board.is_terminal():
            break
        for _ in range(num_rollouts):
            tree.do_rollout(board)
        board = tree.choose(board)

    vc = board.get_cover()
    if display:
        print("\n\nMCTS APPROX\n")
        print(vc)
        color_map = []
        for node in G:
            if node in vc:
                color_map.append('red')
            else:
                color_map.append('blue')
        plt.figure(1)
        nx.draw(G, node_color=color_map, with_labels=True)

    return vc
```

In [5]:
```
def self_play(G, net, num_rollouts=80):
    tree = MCTS_AZ(net)
    board = VertexCoverInstance(G)
    data = []
    while True:
        if board.is_terminal():
            break
        for _ in range(num_rollouts):
            tree.do_rollout(board)
        new_board, policy = tree.choose_by_policy(board)
        record = from_networkx(board.graph)
        record.y = torch.tensor(policy).reshape(1,-1)
        data.append(record)
        board = new_board

    return data
```

In [4]:
```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.utils import from_networkx

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = GCNConv(1, 16)
        self.conv2 = GCNConv(16, 16)
        self.conv_prob = GCNConv(16, 1)

    # expects a torch_geometric Data object
    def forward(self, data):
        num_nodes = data.num_nodes
        x = torch.tensor([[1.0] for _ in range(num_nodes)])
        edge_index = data.edge_index
        x = self.conv1(x ,edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        edge_index = data.edge_index
        x = self.conv2(x ,edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        probs = self.conv_prob(x, edge_index).reshape(-1,num_nodes)
        return F.softmax(probs,dim=1)

    def predict(self, instance):
        data = from_networkx(instance.graph)
        return self.forward(data).flatten().tolist()
```

In [21]:
```
from torch_geometric.data import DataLoader

TRAIN_GRAPH_SIZE = 10
TEST_GRAPH_SIZE = 10

NUM_ITERS = 10
NUM_SELF_GAMES = 100
NUM_EPOCHS = 40
NUM_EVAL_SIMS = 50

scores = []

net = Net()
optimizer = torch.optim.Adam(net.parameters(), lr = 0.0001)
criterion=torch.nn.BCELoss()

for i in range(NUM_ITERS):
    print('Iteration %d' % (i))

    # get self-play records
    print("Getting self-play records")
    records = []
    for i in range(NUM_SELF_GAMES):
        A = generate_graph(TRAIN_GRAPH_SIZE)
        records.extend(self_play(A, net))
        if i % 20 == 19:
            print("  Generated game %d" % (i+1))

    # training
    print("Training")
    data_loader = DataLoader(records, batch_size=32, shuffle=True)
    for epoch in range(NUM_EPOCHS):
        total_loss = 0
        num_batches = 0
        for batch in data_loader:

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(batch)
            loss = criterion(outputs, batch.y)
            total_loss += loss.item()
            num_batches += 1
            loss.backward()
            optimizer.step()

        # print statistics
        if epoch % 10 == 0:
            print('Epoch %d loss: %.3f' %
                  (epoch, total_loss / num_batches))

    # evaluate on larger test graphs
    print("Evaluating")
    count_success = 0
    for x in range(NUM_EVAL_SIMS):
```

```
        A = generate_graph(TEST_GRAPH_SIZE)
        vc = play_game_AZ(A, net,display=False)
        opt = show_optimal_vc(A, display=False)
        if len(vc)<=len(opt)*1.1:
            count_success=count_success+1
    print('Score: %.3f' % (count_success / NUM_EVAL_SIMS))
    scores.append(count_success / NUM_EVAL_SIMS)


plt.plot(scores)
plt.ylabel('Performance')
plt.xlabel('Self Play / Training Iterations')
plt.title('Percentage of Good AlphaMinVertex Approximations during Train
ing')
plt.show()
```

```
Iteration 0
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.570

/opt/anaconda3/lib/python3.7/site-packages/torch/nn/modules/loss.py:49
8: UserWarning: Using a target size (torch.Size([32, 10])) that is diff
erent to the input size (torch.Size([1, 320])) is deprecated. Please en
sure they have the same size.
  return F.binary_cross_entropy(input, target, weight=self.weight, redu
ction=self.reduction)
/opt/anaconda3/lib/python3.7/site-packages/torch/nn/modules/loss.py:49
8: UserWarning: Using a target size (torch.Size([2, 10])) that is diffe
rent to the input size (torch.Size([1, 20])) is deprecated. Please ensu
re they have the same size.
  return F.binary_cross_entropy(input, target, weight=self.weight, redu
ction=self.reduction)

Epoch 10 loss: 0.568
Epoch 20 loss: 0.567
Epoch 30 loss: 0.566
Evaluating
Score: 0.620
Iteration 1
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.580

/opt/anaconda3/lib/python3.7/site-packages/torch/nn/modules/loss.py:49
8: UserWarning: Using a target size (torch.Size([30, 10])) that is diff
erent to the input size (torch.Size([1, 300])) is deprecated. Please en
sure they have the same size.
  return F.binary_cross_entropy(input, target, weight=self.weight, redu
ction=self.reduction)
```

```
Epoch 10 loss: 0.580
Epoch 20 loss: 0.579
Epoch 30 loss: 0.578
Evaluating
Score: 0.580
Iteration 2
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.575
```

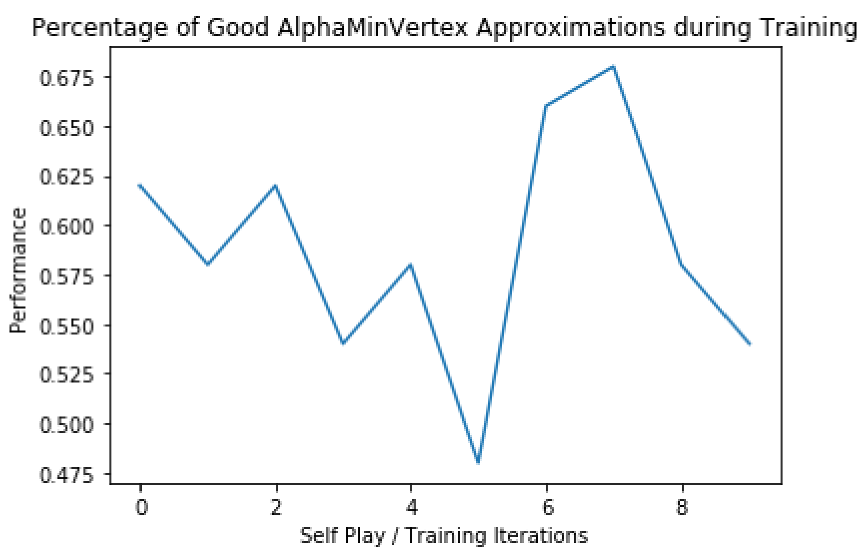
```
/opt/anaconda3/lib/python3.7/site-packages/torch/nn/modules/loss.py:49
8: UserWarning: Using a target size (torch.Size([15, 10])) that is diff
erent to the input size (torch.Size([1, 150])) is deprecated. Please en
sure they have the same size.
  return F.binary_cross_entropy(input, target, weight=self.weight, redu
ction=self.reduction)
```

```
Epoch 10 loss: 0.574
Epoch 20 loss: 0.574
Epoch 30 loss: 0.574
Evaluating
Score: 0.620
Iteration 3
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.579
Epoch 10 loss: 0.579
Epoch 20 loss: 0.579
Epoch 30 loss: 0.578
Evaluating
Score: 0.540
Iteration 4
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.577
```

```
/opt/anaconda3/lib/python3.7/site-packages/torch/nn/modules/loss.py:49
8: UserWarning: Using a target size (torch.Size([26, 10])) that is diff
erent to the input size (torch.Size([1, 260])) is deprecated. Please en
sure they have the same size.
  return F.binary_cross_entropy(input, target, weight=self.weight, redu
ction=self.reduction)
```

```
Epoch 10 loss: 0.576
Epoch 20 loss: 0.576
Epoch 30 loss: 0.576
Evaluating
Score: 0.580
Iteration 5
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.561
Epoch 10 loss: 0.562
Epoch 20 loss: 0.562
Epoch 30 loss: 0.562
Evaluating
Score: 0.480
Iteration 6
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.578
Epoch 10 loss: 0.577
Epoch 20 loss: 0.577
Epoch 30 loss: 0.577
Evaluating
Score: 0.660
Iteration 7
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.577
Epoch 10 loss: 0.577
Epoch 20 loss: 0.577
Epoch 30 loss: 0.577
Evaluating
Score: 0.680
Iteration 8
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.574
```

```
/opt/anaconda3/lib/python3.7/site-packages/torch/nn/modules/loss.py:49
8: UserWarning: Using a target size (torch.Size([17, 10])) that is diff
erent to the input size (torch.Size([1, 170])) is deprecated. Please en
sure they have the same size.
  return F.binary_cross_entropy(input, target, weight=self.weight, redu
ction=self.reduction)

Epoch 10 loss: 0.574
Epoch 20 loss: 0.573
Epoch 30 loss: 0.574
Evaluating
Score: 0.580
Iteration 9
Getting self-play records
  Generated game 20
  Generated game 40
  Generated game 60
  Generated game 80
  Generated game 100
Training
Epoch 0 loss: 0.564
Epoch 10 loss: 0.562
Epoch 20 loss: 0.562
Epoch 30 loss: 0.562
Evaluating
Score: 0.540
```



In [ ]: