procedure brute\_maxcut(G):

Input: Graph G = (V, E) in an adjacency list

Output: Binary array of the coloring of nodes

coloring = [0] \* Length(V) (list of size V, all values start at 0 == ‘white’)

current\_cut = 0

max\_coloring = [0] \* Length(V) (only stores that coloring with the highest result)

max\_cut = 0

while (sum(coloring)) < length(V):

coloring = Increment the rightmost element of the array by 1. (If the number is already 1, then you carry, similar to binary addition.)

changed = index of changed value

Evaluate the state by checking the edges of the node that was just colored.

current\_cut += evaluate\_state(coloring, changed)

If (current\_cut > max\_cut):

max\_cut = current\_cut

max\_coloring = coloring

procedure evaluate\_state(G, coloring, changed):

Input: coloring: Graph G, current binary list, changed: index of changed node

Output: The # of edges being added by the cut, i.e. the # of edges between the sets.

edge\_count = 0

for edge in range(len(G[changed])): (coloring is indexed the same)

check to see if the edges contain alternating colors

if (coloring[changed] != coloring[edge]):

edge\_count += 1

return edge\_count

procedure randomized\_cut(G):

Input: Graph G = (V, E) in adjacency list

Output: list of colorings as binary, # of edges between sets

current\_cut = 0

max\_coloring = [0] \* Length(V) (only stores that coloring with the highest result)

max\_cut = 0

random\_lists = 2^(n / 4) random lists to evaluate

for list in random\_lists:

current\_cut = evaluate(random\_lists)

if (current\_cut > max\_cut):

max\_cut = current\_cut

max\_coloring = list

return max\_colorings, max\_cut

procedure evaluate(G, coloring):

Input: Graph G, coloring: binary list of node colorings, 1 = “black”, 0 = “white”

Output: # of edges between the sets given a binary list

cut\_count = 0

for node in range(len(coloring)):

for edge in range(len(G[node])):

node color != other node color:

cut\_count += 1

return cut\_count

procedure greedy\_one(G):

Input: graph G

Output: list of colorings as binary, # of edges between sets

coloring = [0] \* Length(V) (list of size V, all values start at 0 == ‘white’)

current\_cut = 0

max\_cut = 0

max\_coloring = [0] \* Length(V) (only stores that coloring with the highest result)

priority = [] (list ordered by vertices of highest to lowest degree)

for node in priority:

change node from 0 to 1 in max\_coloring

changed = index of changed value

Evaluate the state by checking the edges of the node that was just colored.

current\_cut += evaluate\_state(coloring, changed)

If (current\_cut > max\_cut):

max\_cut = current\_cut

max\_coloring = coloring

If the node was a bad choice undo, the flip

procedure evaluate\_state(G, coloring, changed):

Input: coloring: Graph G, current binary list, changed: index of changed node

Output: The # of edges being added by the cut, i.e. the # of edges between the sets.

edge\_count = 0

for edge in range(len(G[changed])): (coloring is indexed the same)

check to see if the edges contain alternating colors

if (coloring[changed] != coloring[edge]):

edge\_count += 1

return edge\_count

procedure greedy\_two(G):

Input: graph G

Output: list of colorings as binary, # of edges between sets

coloring = [0] \* Length(V) (list of size V, all values start at 0 == ‘white’)

current\_cut = 0

max\_cut = 0

max\_coloring = [0] \* Length(V) (only stores that coloring with the highest result)

for i in range(len(G) / 2):

choose random node to flip

changed = index of changed value

Evaluate the state by checking the edges of the node that was just colored.

current\_cut += evaluate\_state(coloring, changed)

If (current\_cut > max\_cut):

max\_cut = current\_cut

max\_coloring = coloring

If the node was a bad choice undo, the flip

procedure evaluate\_state(G, coloring, changed):

Input: coloring: Graph G, current binary list, changed: index of changed node

Output: The # of edges being added by the cut, i.e. the # of edges between the sets.

edge\_count = 0

for edge in range(len(G[changed])): (coloring is indexed the same)

check to see if the edges contain alternating colors

if (coloring[changed] != coloring[edge]):

edge\_count += 1

return edge\_count