void Parser::readEC(char\* addr, char\* qAddr, const Para& myPara) {

ipair\_t hdUb(myPara.hdMax, myPara.hdMax);

ipair\_t dPoints(0, myPara.K - myPara.step); //dividing points //a

std::string myRead = addr;

int readLen = myRead.length();

int currPos = 0, prePos = -1;

bool crawlingFlag = false;

int crawlTo = -1;

while (1) {

upair\_t mosaic;

uint64\_t ID1, ID2;

if (!(toID(ID1, &myRead[currPos], myPara.K) &&

toID(ID2, &myRead[currPos + myPara.step], myPara.K))) {

return; /\* containing non-acgt\*/

}

mosaic = upair\_t(ID1, ID2);

if (errorCall(mosaic, dPoints, hdUb, qAddr + currPos, myPara)) {//a

updateRead(myRead, currPos);

if (currPos + myPara.step + myPara.K == readLen) {

return;

}

prePos = currPos;

if (!crawlingFlag) {

if (currPos + myPara.step <= readLen - myPara.K - myPara.step) {

dPoints = ipair\_t(myPara.K, 0);

currPos += myPara.step;

hdUb = ipair\_t(0, myPara.hdMax);

}

else {

dPoints = ipair\_t(myPara.K, currPos + myPara.K + 2 \* myPara.step - readLen);

currPos = readLen - myPara.K - myPara.step;

hdUb = ipair\_t(0, myPara.hdMax);

}

}

}

else {

if (crawlingFlag == false){

crawlingFlag = true;

crawlTo = currPos;

}

}

if (crawlingFlag){

// crawling --- to do reverse crawling

if (currPos == prePos + 1 || prePos == -1){

return;

}

currPos = prePos + 1;

if(currPos == crawlTo) crawlingFlag = false;

hdUb = ipair\_t(0, 1);

dPoints = ipair\_t(myPara.K, myPara.K - 3);

}

}

}

while (1) {

upair\_t mosaic;

uint64\_t ID1, ID2;

if (!(toID(ID1, &myRead[nextPos], myPara.K) &&

toID(ID2, &myRead[nextPos + myPara.step], myPara.K))) {

return; /\* containing non-acgt\*/

}

mosaic = upair\_t(ID1, ID2);

if (errorCall(mosaic, dPoints, hdUb, qAddr + nextPos, myPara)) {//a

updateRead(myRead, nextPos);

if (nextPos + myPara.step + myPara.K == readLen) {

return;

}

prePos = nextPos;

if (crawlingFlag == false) {

if (nextPos + myPara.step <= readLen - myPara.K - myPara.step) {

dPoints = ipair\_t(myPara.K, 0);

nextPos += myPara.step;

hdUb = ipair\_t(0, myPara.hdMax);

}

else {

dPoints = ipair\_t(myPara.K, nextPos + myPara.K + 2 \* myPara.step - readLen);

nextPos = readLen - myPara.K - myPara.step;

hdUb = ipair\_t(0, myPara.hdMax);

}

}

if (crawlingFlag == true){

// crawling --- to do reverse crawling

if (nextPos == prePos + 1 || prePos == -1){

return;

}

nextPos = prePos + 1;

if(nextPos == crawlTo) {

crawlingFlag = false;

}

hdUb = ipair\_t(0, 1);

dPoints = ipair\_t(myPara.K, myPara.K - 3);

}

} else { // else errorCall

if (crawlingFlag == false){

crawlingFlag = true;

crawlTo = nextPos;

}

if (crawlingFlag == true){

// crawling --- to do reverse crawling

if (nextPos == prePos + 1 || prePos == -1){

return;

}

nextPos = prePos + 1;

if(nextPos == crawlTo) {

crawlingFlag = false;

}

hdUb = ipair\_t(0, 1);

dPoints = ipair\_t(myPara.K, myPara.K - 3);

}

} // end if errorCall

} // end while(1)

void Parser::readEC(char\* addr, char\* qAddr, const Para& myPara) {

ipair\_t hdUb(myPara.hdMax, myPara.hdMax);

ipair\_t dPoints(0, myPara.K - myPara.step); //dividing points //a

std::string myRead = addr;

int readLen = myRead.length();

int currPos = 0, prePos = -1;

bool crawlingFlag = false;

int crawlTo = -1;

while (1) {

upair\_t mosaic;

uint64\_t ID1, ID2;

if (!(toID(ID1, &myRead[nextPos], myPara.K) &&

toID(ID2, &myRead[nextPos + myPara.step], myPara.K))) {

break; /\* containing non-acgt\*/

}

mosaic = upair\_t(ID1, ID2);

if (errorCall(mosaic, dPoints, hdUb, qAddr + nextPos, myPara)) {//a

updateRead(myRead, nextPos);

if (nextPos + myPara.step + myPara.K == readLen) {

break;

}

prePos = nextPos;

if (crawlingFlag == false) {

if (nextPos + myPara.step <= readLen - myPara.K - myPara.step) {

dPoints = ipair\_t(myPara.K, 0);

nextPos += myPara.step;

hdUb = ipair\_t(0, myPara.hdMax);

}

else {

dPoints = ipair\_t(myPara.K, nextPos + myPara.K + 2 \* myPara.step - readLen);

nextPos = readLen - myPara.K - myPara.step;

hdUb = ipair\_t(0, myPara.hdMax);

}

} else {

hdUb = ipair\_t(0, 1);

nextPos = prePos + 1;

dPoints = ipair\_t(myPara.K, myPara.K - 3);

}

} else { // else errorCall

if (crawlingFlag == false){

crawlTo = nextPos;

}

if(crawlTo == prePos + 1) {

crawlingFlag = false;

} else {

crawlingFlag = true;

}

// crawling --- to do reverse crawling

if (prePos == -1 || nextPos == prePos + 1){

break;

}

nextPos = prePos + 1;

hdUb = ipair\_t(0, 1);

dPoints = ipair\_t(myPara.K, myPara.K - 3);

} // end if errorCall

}

}

void Parser::mergeTiles(kcvec\_t& tileTo, kcvec\_t& tileFrom, const Para& myPara) {

/\*

\* convert IDs in [tileFrom] to revcompl IDs and insert to [tileTo]

\* then sort and merge (remove duplicates); keep the repID in place

\*/

if (tileFrom.size() == 0) return;

/\* convert to reverse compl IDs

\*/

for (unsigned i = 0; i < tileFrom.size(); ++i) {

tileFrom[i].ID = reverse\_complementary <uint64\_t, uint64\_t >

(tileFrom[i].ID, myPara.K + myPara.step);

}

if (tileTo.size() == 0){

tileTo = tileFrom;

tileFrom.clear();

return;

}

uint64\_t repID = tileTo[0].ID;

tileTo.insert(tileTo.end(), tileFrom.begin(), tileFrom.end());

tileFrom.clear();

/\* sort

\*/

std::sort(tileTo.begin(), tileTo.end(), Knumcomp());

/\* merge

\*/

int idx1 = 0;

for (int idx2 = idx1 + 1; idx2 < (int)tileTo.size(); ++idx2) {

if (tileTo[idx2].ID == tileTo[idx1].ID) {

tileTo[idx1].goodCnt += tileTo[idx2].goodCnt;

tileTo[idx1].cnt += tileTo[idx2].cnt;

} else {

idx1 = idx2;

}

}

kcvec\_t::iterator it =

std::unique\_copy(tileTo.begin(), tileTo.end(), tileTo.begin(), unicpy);

tileTo.resize(it - tileTo.begin());

/\* keep repID in the first pos

\*/

if (tileTo[0].ID != repID) {

for (unsigned i = 1; i < tileTo.size(); ++i) {

if (tileTo[i].ID == repID) {

kc\_t tmp = tileTo[0];

tileTo[0] = tileTo[i];

tileTo[i] = tmp;

break;

}

}

}

}

bool Parser::overlay(uint64\_t& rslt, uint32\_t n1, uint32\_t n2, const Para& myPara) {

int shift = 2 \* myPara.step;

uint64\_t a = n1, b = n2;

// comparison suffix - prefix if necessary

if (myPara.step >= myPara.K ||

(b >> shift) == (a & ((1 << 2\*(myPara.K - myPara.step)) - 1))) {

rslt = (a << shift) | b;

return true;

}

return false;

}

void Parser::tiling(kcvec\_t& tiles, const uvec\_t& N1,

const uvec\_t& N2, const Para& myPara) {

if (N1.size() == 0 || N2.size() == 0) return;

uint64\_t reptile;

if (!overlay(reptile, N1[0], N2[0], myPara)) {

std::cout << "Err: errCall, reptile construction fail\n";

exit(1);

}

kc\_t output;

int idx = ecdata->findTile(reptile,output);

if (idx != -1) {

tiles.push\_back(output);

}

else {

tiles.push\_back(kc\_t(reptile, 0, 0));

}

for (unsigned i = 0; i < N1.size(); ++i) {

for (unsigned j = 0; j < N2.size(); ++j) {

if (i == 0 && j == 0) continue;

uint64\_t tmptile;

if (overlay(tmptile, N1[i], N2[j], myPara)) {

//binary search

kc\_t tmpoutput;

idx = ecdata->findTile(tmptile,tmpoutput);

if (idx != -1) {

tiles.push\_back(tmpoutput);

}

}

}

}

}

bool Parser::errorCall(const upair\_t& mosaic, const ipair\_t& dPoints,

const ipair\_t& hdUb, char\* qAddr, const Para& myPara) {

upair\_t rvMosaic(

reverse\_complementary<uint32\_t, uint32\_t > (mosaic.first, myPara.K),

reverse\_complementary<uint32\_t, uint32\_t > (mosaic.second, myPara.K));

uvec\_t heads, tails, rvHeads, rvTails;

if (ecdata->findKmer((kmer\_id\_t)mosaic.first)){

heads.push\_back(mosaic.first);

}

if (ecdata->findKmer((kmer\_id\_t)mosaic.second)){

tails.push\_back(mosaic.second);

}

if (ecdata->findKmer((kmer\_id\_t)rvMosaic.second)){

rvHeads.push\_back(rvMosaic.second);

}

if (ecdata->findKmer((kmer\_id\_t)rvMosaic.first)){

rvTails.push\_back(rvMosaic.first);

}

ipair\_t hd(0, 0);

while (hd.first <= hdUb.first) {

kcvec\_t tiles, rvTiles;

tiling(tiles, heads, tails, myPara);

tiling(rvTiles, rvHeads, rvTails, myPara);

mergeTiles(tiles, rvTiles, myPara);

if (tiles.size() > 0) { // tiles.size() may equal to 0 due to error correction

if (tiles[0].goodCnt >= myPara.tGoodTile) {

return true;

}

else {

if (tiles.size() == 1) { // try to correct

if (tiles[0].goodCnt >= myPara.tCard &&

goodQuals(qAddr, myPara.K + myPara.step, myPara.Qlb)) {

return true;

}

} else {

ivec\_t candies; // indices of tiles

candidates(candies, tiles, myPara.tCard);

if (tiles[0].goodCnt >= myPara.tCard) {

//

// ec only if non-ambig correction of low qual bases

// could be identified and correcting to one of the

// high cardinality neighbors

//

int tGoodCnt = tiles[0].goodCnt / myPara.tRatio;

ivec\_t highCardNbs;

for (unsigned i = 0; i < candies.size(); ++i) {

if (tiles[candies[i]].goodCnt >= tGoodCnt)

highCardNbs.push\_back(candies[i]);

}

int alterNum = 0;

for (unsigned i = 0; i < highCardNbs.size(); ++i) {

// check all candidates, if ambig, then do not ec

evec\_t errs;

diff(errs, tiles[highCardNbs[i]].ID, tiles[0].ID,

qAddr, myPara.step + myPara.K);

for (unsigned j = 0; j < errs.size(); ++j) {

if (errs[j].qual < myPara.Qlb) {

readErr\_ = errs;

alterNum++;

break;

}

}

}

if (alterNum > 1) { // ambig

readErr\_.clear();

} else return true;

} else {

// ec only if non-ambig correction to one of [candies] could be identified

if (candies.size() == 1) {

diff(readErr\_, tiles[candies[0]].ID, tiles[0].ID,

qAddr, myPara.step + myPara.K);

return true;

}

}

}

}

} // end of tiles.size() > 0

//

// increase Hamming Distance to search for more neighbors

//

hd.second++;

if (hd.second <= hdUb.second) {

unitNeighbor(tails, dPoints.second, myPara);

unitNeighbor(rvHeads, dPoints.second, myPara);

} else {

hd.first++;

if (hd.first <= hdUb.first) {

unitNeighbor(heads, dPoints.first, myPara);

unitNeighbor(rvTails, dPoints.first, myPara);

}

else{ // all possible searches have been done, till this stage

// if tile[0] is the maximum and > tCard then, it is considered

// as correct due to low coverage region

int maxCnt = 0;

bool tflag = true;

if (tiles.size() > 0) maxCnt = tiles[0].goodCnt;

if (maxCnt >= myPara.tCard) {

for (unsigned i = 1; i < tiles.size(); i ++){

if (maxCnt < tiles[i].goodCnt) {

tflag = false;

break;

}

}

if (tflag) return true;

}

}

}

} // End of while loop

return false;

}