Solutions: Genetic Algorithms

created by Lisa Wang

```
/* struct Gene
 * -----------
 * Stores information about a gene in the chromosome, which can be seen as a
 * node in the chromosome linked list.
 */
struct Gene {
    char letter;
    Gene * next;
};

/* struct Chromosome
 * -----------------
 * A wrapper struct for the linked list that stores a pointer to the head of
 * the list as well as the size.
 */
struct Chromosome {
    int size;
    Gene * head;
};

/* Function: initChromosome
 * Usage: initChromosome(child);
 * -----------------------------
 * Initializes a chromosome, by creating a new wrapper struct and initializing
 * the fields in the struct to their default values. Note that chrom has to be
 * passed by reference since we are changing the value of the pointer.
 */
void initChromosome(Chromosome * & chrom) {
    chrom = new Chromosome;
    chrom->size = 0;
    chrom->head = NULL;
}

/* Function: shallowCopy
 * Usage: shallowCopy(parent, child);
 * ----------------------------------
 * Creates a shallow copy of the source linked list and puts the copy into dest.
 * It is shallow since the linked list exists once but the head pointers
 * in both Chromosome structs point to the same head after calling this
 * function.
 */
void shallowCopy(Chromosome * src, Chromosome * dest) {
    dest->size = src->size;
    dest->head = src->head;
}
```

```
/* Function: crossover
 * Usage: int result = crossover(parentA, parentB, childA, childB);
 * ----------------------------------------------------------------
 * Given two parent chromosomes of the same size, this function creates two new child
 * chromosomes by crossing over the parents and deletes the parent wrapper structs.
 * A crossoverPoint of value x specifies that the first x genes from parentA should go
 * to childA and the rest should go to childB.
 * E.g. if parentA = ABC, parentB = DEF and crossoverPoint = 2, then childA = ABF and
 * childB = DEC.This also implies that if crossoverPoint is 0, the children are swapped
 * copies of the parents and if crossoverPoint is the size of the chromosome, the children
 * are exact copies of their parents.
 * Returns -1 if the parent chromosomes do not have the same size and the crossoverPoint
 * when the crossover succeeded.
 * Note that we have to pass in the child chromosome pointers by reference, since this
 * function modifies them.
 */
int crossover(Chromosome * parentA, Chromosome * parentB, Chromosome * & childA,
                              Chromosome * & childB) {
    if (parentA->size != parentB->size) return -1;
    int crossoverPoint = randomInteger(0, parentA->size); // bounds are inclusive
    initChromosome(childA);
    initChromosome(childB);
    if (crossoverPoint == 0) {
        // edge case: childB is a copy of parentA, childA is a copy of parentB
        shallowCopy(parentA, childB);
        shallowCopy(parentB, childA);
        return crossoverPoint;
    }
    shallowCopy(parentA, childA);
    shallowCopy(parentB, childB);
    Gene * curA = childA->head;
    Gene * curB = childB->head;
    for (int i = 0; i < crossoverPoint - 1; i++) {
        curA = curA->next;
        curB = curB->next;
    }
    // swap the remaining portions of the two chromosomes
    Gene * temp = curA->next;
    curA->next = curB->next;
    curB->next = temp;
    // delete the parent wrapper structs
    delete parentA;
    delete parentB;
    return crossoverPoint;
}


/* Function: mutate
 * Usage: mutate(child, geneAlphabet);
 * -----------------------------------
 * Mutates the chromosome by deleting a gene randomly, and inserting a gene randomly.
 * The deletion and the insertion do not have to be at the same position in the
 * chromosome.
 */
```

```
void mutateChromosome(Chromosome * chrom, Vector<char> & geneAlphabet) {
    if (chrom->size == 0) return;
    int deletionIndex = randomDeleteGene(chrom);
    int insertionIndex = randomInsertGene(chrom, geneAlphabet);
}

/* Function: randomDeleteGene
 * Usage: int deletionIndex = randomDeleteGene(child);
 * ---------------------------------------------------
 * Randomly deletes one gene from the chromosome.
 */
int randomDeleteGene(Chromosome * chrom) {
    int deletionIndex = randomInteger(0, chrom->size - 1);
    cout << "deletionIndex: " << deletionIndex << endl;
    Gene * prev = NULL;
    if (deletionIndex == 0) { // If first gene needs to be deleted
        chrom->head = cur->next;
    } else {
        for ( int i = 0; i < deletionIndex; i++) {
            prev = cur;
            cur = cur->next;
        }
        prev->next = cur->next;
    }
    delete cur;
    chrom->size--;
    return deletionIndex;
}

/* Function: randomInsertGene
 * Usage: int insertionIndex = randomInsertGene(child);
 * ----------------------------------------------------
 * Randomly inserts one gene into the chromosome.
 */
int randomInsertGene(Chromosome * chrom, Vector<char> & geneAlphabet) {
    int insertionIndex = randomInteger(0, chrom->size);
    char letter = geneAlphabet[randomInteger(0, geneAlphabet.size()-1)];
    Gene * gene = new Gene;
    gene->letter = letter;
    gene->next = NULL;
    Gene * cur = chrom->head;
    if (insertionIndex == 0) { // If new gene needs to be inserted at the beginning
        gene->next = chrom->head;
        chrom->head = gene;
    } else {
        for (int i = 1; i < insertionIndex; i++) cur = cur->next;
        Gene * temp = cur->next;
        cur->next = gene;
        gene->next = temp;
    }
    chrom->size++;
    return insertionIndex;
}
```