2nd Lab Class – Backtracking (*Algoritmos de Retrocesso*) – Solutions

---

Note: Keep in mind that there may be several possible solutions to these problems. The solutions herein presented are suggested as possible alternatives.

---

1. Labyrinth

a.

```cpp
bool Labirinth::findGoal(int x, int y) {
   initializeVisited();
   return findGoalRec(x,y);
}

// Auxiliary recursive function
bool Labirinth::findGoalRec(int x, int y) {
   // Check if this position is worth visiting (limits checking could
   // be omitted because the labyrinth is surrounded by walls)
   if (x < 0 || y < 0 || x >= 10 || x >= 10
      || labirinth[y][x] == 0 || visited[y][x])
         return false;

   // Mark as visited
   visited[y][x] = true;

   // Check if the exit was reached
   if (labirinth[y][x] == 2) {
      cout << ": Reached the goal!" << endl;
      return true;
   }

   // Try all the adjacent cells
   return findGoalRec(x-1, y) || findGoalRec(x+1, y)
        || findGoalRec(x, y-1) || findGoalRec(x, y+1);
}
```

b. $T(n) = O(n^2)$ in the worst case, where n represents the dimension of the labyrinth (on this case n=10) since each cell is visited at most once.

2. Sudoku

a.

```cpp
/**
 * Solves the Sudoku, that is, fills in on all the empty cells,
   satisfying the Sudoku constraints.
 * Returns true if succeeded and false otherwise.
 * Follows a greedy algorithm with backtracking.
 */
```

```
bool Sudoku::solve() {
   if (isComplete())
      return true; // success, terminate

   // Greedy approach: searches the best cell to fill in
   // (with a minimum number of candidates)
   int i, j;
   if ( ! findBestCell(i, j) )
      return false; // impossible, backtrack

   // Tries all the possible candidates in the chosen cell
   for (int n = 1; n <= 9; n++)
      if (accepts(i, j, n)) {
         place(i, j, n);
         if (solve())
          return true; // success, terminate
         clear(i, j);
      }

   return false; // impossible, backtrack
}

/**
 * Searches the best cell to fill in - the cell with
 * a minimum number of candidates.
 * Returns true if found and false otherwise (Sudoku impossible).
 */
bool Sudoku::findBestCell(int & best_i, int & best_j) {
   best_i = -1, best_j = -1;
   int best_num_choices = 10; // above maximum

   for (int i = 0; i < 9 ; i++)
      for (int j = 0; j < 9; j++)
         if (numbers[i][j] == 0) {
            int num_choices = 0;
            for (int n = 1; n <= 9; n++)
               if (accepts(i, j, n))
                  num_choices++;

            if (num_choices == 0)
               return false; // impossible

            if (num_choices < best_num_choices) {
               best_num_choices = num_choices;
               best_i = i;
               best_j = j;
               if (num_choices == 1) // cannot improve
                  return true;
            }
         }

   return best_i >= 0;
}

/**
 * Checks if the cell at line i, column j accepts number n
 */
bool Sudoku::accepts(int i, int j, int n) {
   return !lineHasNumber[i][n]
          && !columnHasNumber[j][n]
          && !block3x3HasNumber[i / 3][j / 3][n];
}
```

```
/**
 * Fills in the cell at line i, column j with number n.
 * Also updates the cell counter.
 */
void Sudoku::place(int i, int j, int n) {
   numbers[i][j] = n;
   lineHasNumber[i][n] = true;
   columnHasNumber[j][n] = true;
   block3x3HasNumber[i / 3][j / 3][n] = true;
   countFilled++;
}

/**
 * Clears the cell at line i, column j.
 * Also updates the cell counter.
 */
void Sudoku::clear(int i, int j) {
   numbers[i][j] = 0;
   lineHasNumber[i][n] = false;
   columnHasNumber[j][n] = false;
   block3x3HasNumber[i / 3][j / 3][n] = false;
   countFilled--;
}

/**
 * Checks if the Sudoku is completely solved.
 */
bool Sudoku::isComplete() {
   return countFilled == 9 * 9;
}
```

b.

```
/**
 * Brute force algorithm.
 * Time: O(n^3). Space: O(n)
 */
int Sudoku::countSolutions() {
    // Step 1: Check if the puzzle is complete
    if (isComplete())
        return 1;

    // Step 2: Select the 'best' cell to fill in (with a minimum number of candidates)
    int best_i, best_j;
    if (!findBestCell(best_i, best_j))
        return 0;

    // Step 3: Test each candidate on the chosen cell
    int count = 0;
    for (int n = 1; n <= 9; n++)
        if (accepts(best_i, best_j, n)) {
            place(best_i, best_j, n);
            count += countSolutions();
            clear(best_i, best_j);
            if (count > 1)
                break;
        }
    return count;
}
```

c.

```
void Sudoku::generate() {
    clear();
    completeProblem();
    reduce();
}

bool Sudoku::completeProblem() {
    // Gradually fills random cells until there is a unique solution.
    // At each step, it checked the puzzle is still solvable.
    int countSolut = countSolutions();
    if (countSolut == 0)
        return false; // impossible, need to reduce

    bool changed  = false;
    while (countSolut > 1) {
        int i = (int) (rand() % 9);
        int j = (int) (rand() % 9);
        int n = 1 + (int) (rand() * 9);

        if (numbers[i][j] == 0 && accepts(i, j, n)) {
            place(i, j, n);
            int count = countSolutions();

            if (count == 0)
                clear(i, j);

            else {
                countSolut = count;
                changed= true;
            }
        }
    }

    return changed;

}

bool Sudoku::reduce() {
    bool changed = false;
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            if (numbers[i][j] != 0) {
                int n = numbers[i][j];
                clear(i, j);
                int c = countSolutions();

                if (c > 1)
                    place(i, j, n);
                else // 0 ou 1

                    changed = true;
            }

    return changed;

}
```

3. Changing making problem (backtracking)

```
bool changeMakingBFRec(unsigned int C[], unsigned int Stock[], unsigned int n,
unsigned int curIndex, unsigned int T, unsigned int curNCoins, unsigned int
curCoins[], unsigned int &minCoins, unsigned int bestCoins[]) {

    if(curIndex == n) {
        if(T == 0) {
            if(curNCoins < minCoins) {
                minCoins = curNCoins;
                // Copy the current state to the array storing the best state found so
far
                for(unsigned int i = 0; i < n; i++) {
                    bestCoins[i] = curCoins[i];
                }
            }
            return true;
        }
        return false;
    }

    // Try including another coin with designation equal to C[n-1]
    bool foundSolWithCoin = false;
    if(curCoins[curIndex] < Stock[curIndex] && T >= C[curIndex]) {
        curCoins[curIndex]++;

        foundSolWithCoin = changeMakingBFRec(C,Stock,n,curIndex,T -
C[curIndex],curNCoins+1,curCoins,minCoins,bestCoins);

        curCoins[curIndex]--; // Don't forget to undo the last choice point!
    }

    // Don't try including another coin
    bool foundSolWithoutCoin =
changeMakingBFRec(C,Stock,n,curIndex+1,T,curNCoins,curCoins,minCoins,bestCoins);

    return foundSolWithCoin || foundSolWithoutCoin;

}

// Time: O(2^(D*S)), Space: O(D*S), where D is the number of coin values and S is the
maximum amount of stock of any value
bool changeMakingBacktracking(unsigned int C[], unsigned int Stock[], unsigned int n,
unsigned int T, unsigned int usedCoins[]) {

    unsigned int minCoins = 0;
    unsigned int curCoins[10000]; // static memory allocation is faster :)

    // curCoins stores the current state, usedCoins (= bestCoins stores) the best
solution found at a given time
    for(unsigned int i = 0; i < n; i++) {
        curCoins[i] = 0;
        minCoins += Stock[i];
    }

    return changeMakingBFRec(C,Stock,n,0,T,0,curCoins,minCoins,usedCoins);
}
```

4. Activity selection (backtracking)

```
void activitySelectionBacktrackingRec(std::vector<Activity> A, std::vector<Activity>
curSolution, std::vector<Activity> &bestSolution) {

    if(A.empty()) {
        if(curSolution.size() > bestSolution.size()) {
            bestSolution = curSolution;
        }
        return;
    }

    // Get the next activity
    Activity nextAct = A.at(A.size() - 1);
    A.pop_back();

    // Try including the next activity
    bool overlapsOne = false;
    for(const auto act: curSolution) {
        if(nextAct.overlaps(act)) {
            overlapsOne = true;
            break;
        }
    }

    if(!overlapsOne) {
        std::vector<Activity> nextSolution = curSolution;
        nextSolution.push_back(nextAct);
        activitySelectionBacktrackingRec(A, nextSolution, bestSolution);
    }

    // Don't try including the next activity
    activitySelectionBacktrackingRec(A, curSolution, bestSolution);
}

std::vector<Activity> activitySelectionBacktracking(std::vector<Activity> A) {
    std::vector<Activity> curSol;
    std::vector<Activity> V;
    activitySelectionBacktrackingRec(A, curSol, V);
    return V;
}
```