MWC Experiment 3

Shivani Bhat

BE EXTC A
2022200013

Aim:

To simulate the dynamic channel assignment technique

Procedure:

1. **System Initialization**:

- The main function starts by seeding the random number generator using srand(time(NULL)).� **User Input for System Parameters**:
- The program prompts the user to enter the **total number of channels** (50-100). The getTotalChannels() function handles this input with validation, ensuring the number is within the valid range.
- It then requests the **cluster size**. The getValidClusterSize() function validates this input against the cellular system formula N=i2+j2+i∗j. It repeatedly prompts the user until a valid size is provided.

2. **Channel Calculation and Distribution**:

- The total channels are divided into two categories: **control channels** (10% of the total, rounded up) and **voice/data channels** (the remaining channels).
- The getTrafficDemand() function prompts the user to enter the specific traffic demand (number of channels needed) for each individual cell in the cluster.

3. **Channel Allocation Process**:

- The allocateChannels() function orchestrates the allocation process.
- It first analyzes the overall demand by calling displayClusterFairness(), which provides a summary of total, average, and maximum demand across the cells.
- **Control Channel Allocation**: displayControlChannelMatrix() creates and displays a matrix showing how control channels are evenly distributed among the cells.

- **Voice Channel Allocation**: This is the core of the program. The displayVoiceChannelAllocation() function manages this process, which includes:
    - **Priority Assignment**: The generateChannelsWithPriority() function assigns a random priority (1-5) to each voice channel, with a pre-set ratio of low-priority (1-2) to high-priority (3-5) channels.
    - **Sorted Allocation**: The program then sorts the channels by priority (highest first) and uses a **round-robin allocation strategy**. This method ensures a fair distribution of the most valuable high-priority channels among all the cells with demand.
    - **Allocation Matrix**: The allocateTrafficChannels() function fills a **traffic matrix**, which maps which voice channels are allocated to which cell.

4. **Performance and Fairness Analysis**:
    - After allocation, the program displays the results in detail:
        - displayFairnessDistribution() presents a table showing how many high and low-priority channels were allocated to each cell.
        - displayTrafficMatrix() provides a clear, row-by-row view of the specific channel IDs assigned to each cell.
        - displaySatisfactionMatrix() gives a final performance summary. It calculates and displays the **satisfaction percentage** (demand met) and the **blocking percentage** (demand not met) for each cell and for the system as a whole.
5. **Memory Management**:
    - The program uses dynamic memory allocation for the matrices. At the end of the allocateChannels() and displayVoiceChannelAllocation() functions, all allocated memory is properly freed to prevent leaks.
    - The main function returns 0 upon completion, signaling a successful execution.

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <stdbool.h>


#define MAX_CHANNELS 100
```

```
#define MAX_CLUSTER_SIZE 50


typedef struct {

    int channelId;

    int priority;

} ChannelInfo;


// Function prototypes

int getTotalChannels();

int getValidClusterSize();

bool isValidClusterSize(int N);

void getTrafficDemand(int clusterSize, int* trafficDemand);

void allocateChannels(int clusterSize, int controlChannels, int voiceChannels, int* trafficDemand);

void displayControlChannelMatrix(int clusterSize, int controlChannels);

void displayClusterFairness(int clusterSize, int* trafficDemand);

void normalizeChannelDemand(int clusterSize, int* trafficDemand, int totalAvailableChannels);

void displayVoiceChannelAllocation(int clusterSize, int voiceChannels, int* trafficDemand);

void generateChannelsWithPriority(int voiceChannels, ChannelInfo* channels);

void displayChannelPriorities(ChannelInfo* channels, int voiceChannels);

void allocateTrafficChannels(int clusterSize, ChannelInfo* channels, int voiceChannels, int* trafficDemand, int**
trafficMatrix, int maxCols);

void displayTrafficMatrix(int** trafficMatrix, int clusterSize, int* trafficDemand);

void displaySatisfactionMatrix(int clusterSize, int* trafficDemand, int** trafficMatrix);

void displayFairnessDistribution(int clusterSize, ChannelInfo* channels, int voiceChannels, int* trafficDemand, int**
trafficMatrix, int maxCols);

void shuffleArray(int* array, int size);

int compareChannels(const void* a, const void* b);

int compareInts(const void* a, const void* b);

int findMax(int* array, int size);

int sumArray(int* array, int size);

int countAllocatedChannels(int* row, int maxCols);


int main() {

    srand(time(NULL)); // Initialize random seed
    printf("=== Cellular Channel Allocation System ===\n\n");

    int totalChannels = getTotalChannels();
    int clusterSize = getValidClusterSize();
    int controlChannels = (totalChannels + 9) / 10; // Integer ceiling division for 10%
    int voiceChannels = totalChannels - controlChannels;

    printf("\n=== Channel Distribution ===\n");
    printf("Total Channels: %d\n", totalChannels);
    printf("Control Channels (10%%): %d\n", controlChannels);
    printf("Voice/Data Channels: %d\n", voiceChannels);

    int trafficDemand[MAX_CLUSTER_SIZE];
    getTrafficDemand(clusterSize, trafficDemand);
    allocateChannels(clusterSize, controlChannels, voiceChannels, trafficDemand);

    return 0;
}

int getTotalChannels() {
    int channels;
    do {
        printf("Enter the number of total channels (50-100): ");
        if (scanf("%d", &channels) != 1) {
            printf("Invalid input! Please enter a number.\n");
            // Clear input buffer
            int c;
            while ((c = getchar()) != '\n' && c != EOF);
            continue;
        }
        if (channels < 50 || channels > 100) {
            printf("Invalid input! Total channels must be between 50 and 100.\n");
        }
    } while (channels < 50 || channels > 100);
    return channels;
}

int getValidClusterSize() {
    int clusterSize;
    do {
        printf("Enter the cluster size: ");
        if (scanf("%d", &clusterSize) != 1) {
            printf("Invalid input! Please enter a number.\n");
            // Clear input buffer
            int c;
            while ((c = getchar()) != '\n' && c != EOF);
            continue;
        }
        if (clusterSize <= 0 || clusterSize > MAX_CLUSTER_SIZE) {
            printf("Cluster size must be between 1 and %d.\n", MAX_CLUSTER_SIZE);
            continue;
        }
        if (isValidClusterSize(clusterSize)) {
            // Print the success message only once here
            for (int i = 0; i * i <= clusterSize; i++) {
                for (int j = 0; j * j <= clusterSize; j++) {
                    if (i * i + j * j + i * j == clusterSize) {
                        printf("Correct Cluster Size! Value of i = %d and value of j = %d.\n", i, j);
                        return clusterSize;
                    }
                }
            }
        } else {
            printf("Invalid cluster size. Valid cluster sizes follow the pattern N = i² + j² + i*j\n");
            printf("Common valid sizes: 1, 3, 4, 7, 9, 12, 13, 16, 19, 21, 25, 27, 28, 31...\n");
        }
    } while (true);
    return clusterSize;
}

bool isValidClusterSize(int N) {
    if (N <= 0) return false;

    for (int i = 0; i * i <= N; i++) {
        for (int j = 0; j * j <= N; j++) {
            if (i * i + j * j + i * j == N) {
                return true;
            }
        }
    }
    return false;
}

void getTrafficDemand(int clusterSize, int* trafficDemand) {
    printf("\nEnter traffic channel demand for each cell:\n");
    for (int i = 0; i < clusterSize; i++) {
        do {
            printf("Enter the number of channels for cell %d: ", i + 1);
            if (scanf("%d", &trafficDemand[i]) != 1) {
                printf("Invalid input! Please enter a number.\n");
                // Clear input buffer
                int c;
                while ((c = getchar()) != '\n' && c != EOF);
                continue;
            }
            if (trafficDemand[i] < 0) {
                printf("Channel demand cannot be negative. Please enter a non-negative number.\n");
            }
        } while (trafficDemand[i] < 0);
    }
}
```

```
void allocateChannels(int clusterSize, int controlChannels, int voiceChannels, int* trafficDemand) {
    int totalDemand = sumArray(trafficDemand, clusterSize);

    printf("\n=== Channel Allocation Results ===\n");
    printf("Total traffic demand: %d channels\n", totalDemand);

    if (totalDemand > voiceChannels) {
        printf("Warning: Total demand (%d) exceeds available voice channels (%d)\n", totalDemand, voiceChannels);
        printf("Some channels will be blocked.\n");
    }

    // Display cluster fairness analysis
    displayClusterFairness(clusterSize, trafficDemand);

    // Display control channel matrix BEFORE traffic channels
    displayControlChannelMatrix(clusterSize, controlChannels);

    // Display voice channel allocation
    displayVoiceChannelAllocation(clusterSize, voiceChannels, trafficDemand);
}

void displayControlChannelMatrix(int clusterSize, int controlChannels) {
    printf("\n=== Control Channel Allocation Matrix ===\n");
    printf("Total control channels: %d\n", controlChannels);
    printf("Distribution across %d cells:\n\n", clusterSize);

    int baseChannelsPerCell = controlChannels / clusterSize;
    int extraChannels = controlChannels % clusterSize;

    // Create control channel matrix
    int** controlMatrix = (int**)malloc(clusterSize * sizeof(int*));
    if (controlMatrix == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }


    int channelId = 1;
    for (int i = 0; i < clusterSize; i++) {
        int channelsForCell = baseChannelsPerCell + (i < extraChannels ? 1 : 0);
        controlMatrix[i] = (int*)calloc(channelsForCell, sizeof(int));
        if (controlMatrix[i] == NULL) {
            printf("Memory allocation failed!\n");
            // Free previously allocated memory
            for (int j = 0; j < i; j++) {
                free(controlMatrix[j]);
            }
            free(controlMatrix);
            return;
        }

        // Assign channel IDs
        for (int j = 0; j < channelsForCell; j++) {
            controlMatrix[i][j] = channelId++;
        }
    }


    // Display the matrix
    printf("Control Channel Assignment Matrix:\n");
    for (int i = 0; i < clusterSize; i++) {
        int channelsForCell = baseChannelsPerCell + (i < extraChannels ? 1 : 0);
        printf("Cell %2d: [", i + 1);
        for (int j = 0; j < channelsForCell; j++) {
            if (j > 0) printf(", ");
            printf("%2d", controlMatrix[i][j]);
        }
        printf("]\n");
    }
```

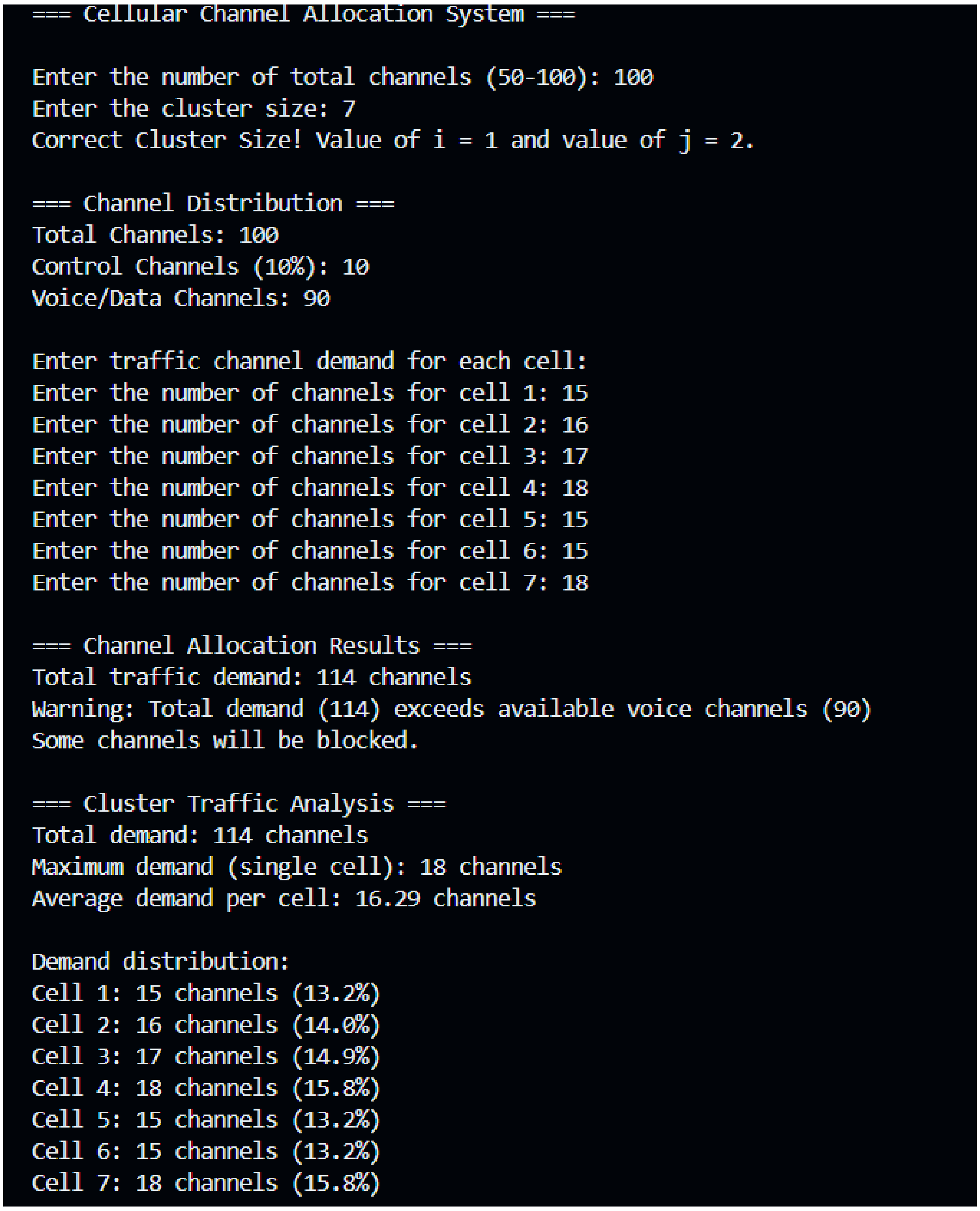
```
    // Free allocated memory
    for (int i = 0; i < clusterSize; i++) {
        free(controlMatrix[i]);
    }
    free(controlMatrix);
}

void displayClusterFairness(int clusterSize, int* trafficDemand) {
    printf("\n=== Cluster Traffic Analysis ===\n");
    int totalDemand = sumArray(trafficDemand, clusterSize);
    int maxDemand = findMax(trafficDemand, clusterSize);
    double avgDemand = (double)totalDemand / clusterSize;

    printf("Total demand: %d channels\n", totalDemand);
    printf("Maximum demand (single cell): %d channels\n", maxDemand);
    printf("Average demand per cell: %.2f channels\n", avgDemand);

    printf("\nDemand distribution:\n");
    for (int i = 0; i < clusterSize; i++) {
        double percentage = (totalDemand > 0) ? (trafficDemand[i] * 100.0 / totalDemand) : 0.0;
        printf("Cell %d: %d channels (%.1f%%)\n", i + 1, trafficDemand[i], percentage);
    }
}

void normalizeChannelDemand(int clusterSize, int* trafficDemand, int totalAvailableChannels) {
    // This function could be used to normalize demands if total exceeds available
    // For now, we'll keep the original demands for allocation
    printf("\nNote: Original traffic demands will be used for allocation.\n");
    printf("Channels will be allocated based on priority and availability.\n");
}

void displayVoiceChannelAllocation(int clusterSize, int voiceChannels, int* trafficDemand) {
    printf("\n=== Voice Channel Allocation ===\n");
```

```
if (voiceChannels <= 0) {
    printf("No voice channels available for allocation.\n");
    return;
}

ChannelInfo channels[MAX_CHANNELS];
generateChannelsWithPriority(voiceChannels, channels);
displayChannelPriorities(channels, voiceChannels);

int maxCols = findMax(trafficDemand, clusterSize);
if (maxCols == 0) {
    printf("No traffic demand from any cell.\n");
    return;
}

// Allocate memory for traffic matrix
int** trafficMatrix = (int**)malloc(clusterSize * sizeof(int*));
if (trafficMatrix == NULL) {
    printf("Memory allocation failed!\n");
    return;
}

for (int i = 0; i < clusterSize; i++) {
    trafficMatrix[i] = (int*)calloc(maxCols, sizeof(int));
    if (trafficMatrix[i] == NULL) {
        printf("Memory allocation failed!\n");
        // Free previously allocated memory
        for (int j = 0; j < i; j++) {
            free(trafficMatrix[j]);
        }
        free(trafficMatrix);
        return;
    }
    }

    allocateTrafficChannels(clusterSize, channels, voiceChannels, trafficDemand, trafficMatrix, maxCols);

    // Display fairness distribution with high/low priority breakdown
    displayFairnessDistribution(clusterSize, channels, voiceChannels, trafficDemand, trafficMatrix, maxCols);

    displayTrafficMatrix(trafficMatrix, clusterSize, trafficDemand);

    // Display additional metrics
    displaySatisfactionMatrix(clusterSize, trafficDemand, trafficMatrix);

    // Free allocated memory
    for (int i = 0; i < clusterSize; i++) {
        free(trafficMatrix[i]);
    }
    free(trafficMatrix);
}

void generateChannelsWithPriority(int voiceChannels, ChannelInfo* channels) {
    int channelNumbers[MAX_CHANNELS];
    for (int i = 0; i < voiceChannels; i++) {
        channelNumbers[i] = i + 1;
    }

    shuffleArray(channelNumbers, voiceChannels);

    int lowPriorityCount = (voiceChannels * 35 + 99) / 100; // Integer ceiling for 35%
    int highPriorityCount = voiceChannels - lowPriorityCount;

    // Assign low priority channels (priority 1-2)
    for (int i = 0; i < lowPriorityCount; i++) {
        channels[i].channelId = channelNumbers[i];
        channels[i].priority = (rand() % 2) + 1;
    }

    // Assign high priority channels (priority 3-5)
    for (int i = lowPriorityCount; i < voiceChannels; i++) {
        channels[i].channelId = channelNumbers[i];
        channels[i].priority = (rand() % 3) + 3;
    }
}

void displayChannelPriorities(ChannelInfo* channels, int voiceChannels) {
    int lowPriority[MAX_CHANNELS], highPriority[MAX_CHANNELS];
    int lowCount = 0, highCount = 0;

    for (int i = 0; i < voiceChannels; i++) {
        if (channels[i].priority <= 2) {
            lowPriority[lowCount++] = channels[i].channelId;
        } else {
            highPriority[highCount++] = channels[i].channelId;
        }
    }

    // Sort arrays using the compareInts function
    qsort(lowPriority, lowCount, sizeof(int), compareInts);
    qsort(highPriority, highCount, sizeof(int), compareInts);

    printf("\nChannel Priority Assignment:\n");
    printf("Low Priority Channels (Priority 1-2): %d channels\n", lowCount);
    if (lowCount > 0) {
        printf("Low Priority: ");
        for (int i = 0; i < lowCount; i++) {
            printf("%d ", lowPriority[i]);
        }
        printf("\n");
    }
    printf("High Priority Channels (Priority 3-5): %d channels\n", highCount);

    if (highCount > 0) {

        printf("High Priority: ");

        for (int i = 0; i < highCount; i++) {

            printf("%d ", highPriority[i]);

        }

        printf("\n");

    }

}


void allocateTrafficChannels(int clusterSize, ChannelInfo* channels, int voiceChannels, int* trafficDemand, int**
trafficMatrix, int maxCols) {

    // Sort channels by priority (highest first)

    qsort(channels, voiceChannels, sizeof(ChannelInfo), compareChannels);


    printf("\n=== Fairness Distribution Process ===\n");

    printf("Using priority-based round-robin allocation\n");


    int channelIndex = 0;

    for (int col = 0; col < maxCols && channelIndex < voiceChannels; col++) {

        for (int cell = 0; cell < clusterSize && channelIndex < voiceChannels; cell++) {

            if (col < trafficDemand[cell]) {

                trafficMatrix[cell][col] = channels[channelIndex].channelId;

                channelIndex++;

            }

        }

    }


    printf("Allocated %d out of %d available voice channels\n", channelIndex, voiceChannels);

}


void displayFairnessDistribution(int clusterSize, ChannelInfo* channels, int voiceChannels, int* trafficDemand, int**
trafficMatrix, int maxCols) {
```

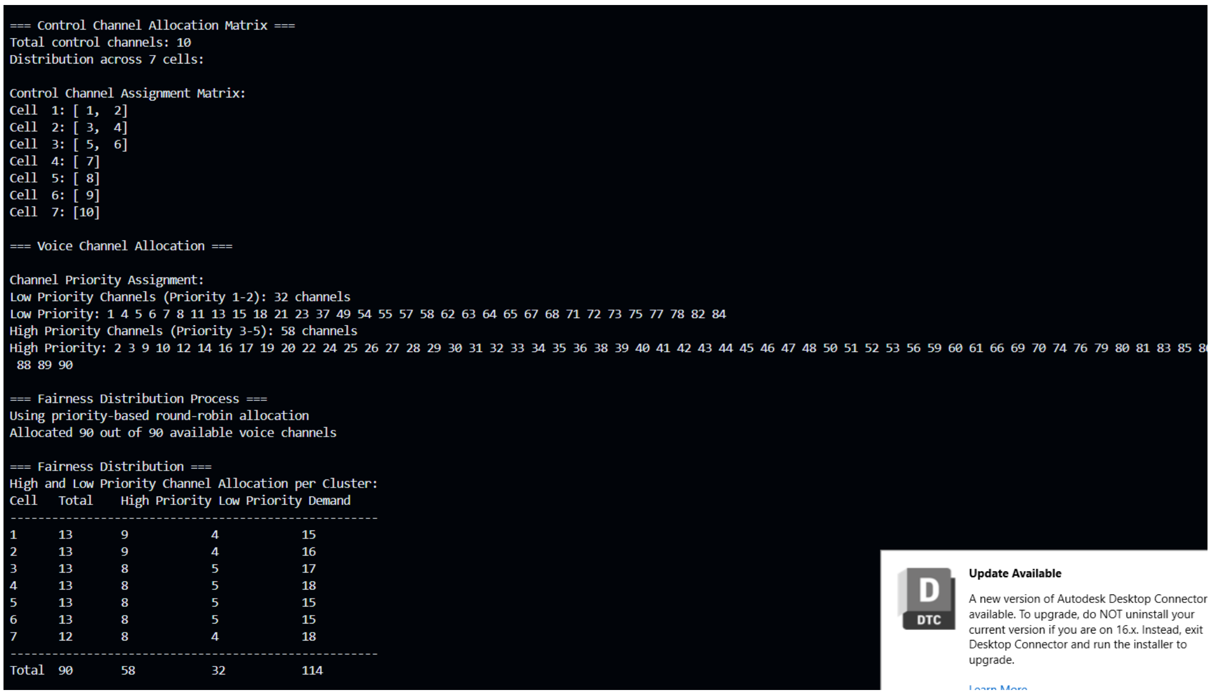
```
printf("\n=== Fairness Distribution ===\n");
printf("High and Low Priority Channel Allocation per Cluster:\n");
printf("%-6s %-8s %-12s %-12s %-10s\n",
     "Cell", "Total", "High Priority", "Low Priority", "Demand");
printf("-----------------------------------------------------\n");

// Create a mapping of channel ID to priority for quick lookup
int channelPriorityMap[MAX_CHANNELS + 1]; // +1 because channel IDs start from 1
for (int i = 0; i <= MAX_CHANNELS; i++) {
   channelPriorityMap[i] = 0;
}
for (int i = 0; i < voiceChannels; i++) {
   channelPriorityMap[channels[i].channelId] = channels[i].priority;
}

int totalHighPriority = 0, totalLowPriority = 0, totalDemand = 0;

for (int cell = 0; cell < clusterSize; cell++) {
   int highPriorityCount = 0, lowPriorityCount = 0;
   int totalAllocated = 0;

   for (int col = 0; col < trafficDemand[cell]; col++) {
      if (trafficMatrix[cell][col] != 0) {
         totalAllocated++;
         int priority = channelPriorityMap[trafficMatrix[cell][col]];
         if (priority <= 2) {
            lowPriorityCount++;
         } else {
            highPriorityCount++;
         }
      }
   }

   printf("%-6d %-8d %-12d %-12d %-10d\n",
            cell + 1, totalAllocated, highPriorityCount, lowPriorityCount, trafficDemand[cell]);

        totalHighPriority += highPriorityCount;
        totalLowPriority += lowPriorityCount;
        totalDemand += trafficDemand[cell];
    }

    printf("----------------------------------------------------\n");
    printf("%-6s %-8d %-12d %-12d %-10d\n",
          "Total", totalHighPriority + totalLowPriority, totalHighPriority, totalLowPriority, totalDemand);
}

void displayTrafficMatrix(int** trafficMatrix, int clusterSize, int* trafficDemand) {
    printf("\n=== Traffic Channel Allocation Matrix ===\n");
    printf("(Channels allocated to each cell)\n");

    for (int row = 0; row < clusterSize; row++) {
        printf("Cell %2d: [", row + 1);
        bool first = true;
        for (int col = 0; col < trafficDemand[row]; col++) {
            if (trafficMatrix[row][col] != 0) {
                if (!first) printf(", ");
                printf("%2d", trafficMatrix[row][col]);
                first = false;
            }
        }
        printf("]\n");
    }
}

// Utility functions
void shuffleArray(int* array, int size) {
    for (int i = size - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        int temp = array[i];

        array[i] = array[j];

        array[j] = temp;

    }

}


int compareChannels(const void* a, const void* b) {

    ChannelInfo* channelA = (ChannelInfo*)a;

    ChannelInfo* channelB = (ChannelInfo*)b;

    return channelB->priority - channelA->priority; // Sort in descending order

}


int compareInts(const void* a, const void* b) {

    return (*(int*)a - *(int*)b); // Sort in ascending order

}


int findMax(int* array, int size) {

    if (size <= 0) return 0;

    int max = array[0];

    for (int i = 1; i < size; i++) {

        if (array[i] > max) {

            max = array[i];

        }

    }

    return max;

}


int sumArray(int* array, int size) {

    int sum = 0;

    for (int i = 0; i < size; i++) {

        sum += array[i];

    }

    return sum;

}
```

```
int countAllocatedChannels(int* row, int maxCols) {
    int count = 0;
    for (int i = 0; i < maxCols; i++) {
        if (row[i] != 0) {
            count++;
        }
    }
    return count;
}

void displaySatisfactionMatrix(int clusterSize, int* trafficDemand, int** trafficMatrix) {
    printf("\n=== Performance Analysis ===\n");
    printf("Demand vs Allocation Summary:\n");
    printf("%-6s %-8s %-10s %-15s %-8s %-10s\n",
           "Cell", "Demand", "Allocated", "Satisfaction%", "Blocked", "Block%");
    printf("--------------------------------------------------------------------\n");

    int totalDemand = 0, totalAllocated = 0, totalBlocked = 0;

    for (int i = 0; i < clusterSize; i++) {
        int allocated = countAllocatedChannels(trafficMatrix[i], trafficDemand[i]);
        int blocked = trafficDemand[i] - allocated;
        double satisfaction = (trafficDemand[i] > 0) ? (allocated * 100.0 / trafficDemand[i]) : 100.0;
        double blockingPercentage = (trafficDemand[i] > 0) ? (blocked * 100.0 / trafficDemand[i]) : 0.0;

        printf("%-6d %-8d %-10d %-15.1f %-8d %-10.1f\n",
               i + 1, trafficDemand[i], allocated, satisfaction, blocked, blockingPercentage);

        totalDemand += trafficDemand[i];
        totalAllocated += allocated;
        totalBlocked += blocked;
    }
    printf("---------------------------------------------------------------------\n");

    double overallSatisfaction = (totalDemand > 0) ? (totalAllocated * 100.0 / totalDemand) : 100.0;

    double overallBlocking = (totalDemand > 0) ? (totalBlocked * 100.0 / totalDemand) : 0.0;


    printf("%-6s %-8d %-10d %-15.1f %-8d %-10.1f\n",

         "Total", totalDemand, totalAllocated, overallSatisfaction, totalBlocked, overallBlocking);

}
```

Output:



```
=== Cellular Channel Allocation System ===

Enter the number of total channels (50-100): 100
Enter the cluster size: 7
Correct Cluster Size! Value of i = 1 and value of j = 2.

=== Channel Distribution ===
Total Channels: 100
Control Channels (10%): 10
Voice/Data Channels: 90

Enter traffic channel demand for each cell:
Enter the number of channels for cell 1: 15
Enter the number of channels for cell 2: 16
Enter the number of channels for cell 3: 17
Enter the number of channels for cell 4: 18
Enter the number of channels for cell 5: 15
Enter the number of channels for cell 6: 15
Enter the number of channels for cell 7: 18

=== Channel Allocation Results ===
Total traffic demand: 114 channels
Warning: Total demand (114) exceeds available voice channels (90)
Some channels will be blocked.

=== Cluster Traffic Analysis ===
Total demand: 114 channels
Maximum demand (single cell): 18 channels
Average demand per cell: 16.29 channels

Demand distribution:
Cell 1: 15 channels (13.2%)
Cell 2: 16 channels (14.0%)
Cell 3: 17 channels (14.9%)
Cell 4: 18 channels (15.8%)
Cell 5: 15 channels (13.2%)
Cell 6: 15 channels (13.2%)
Cell 7: 18 channels (15.8%)
```

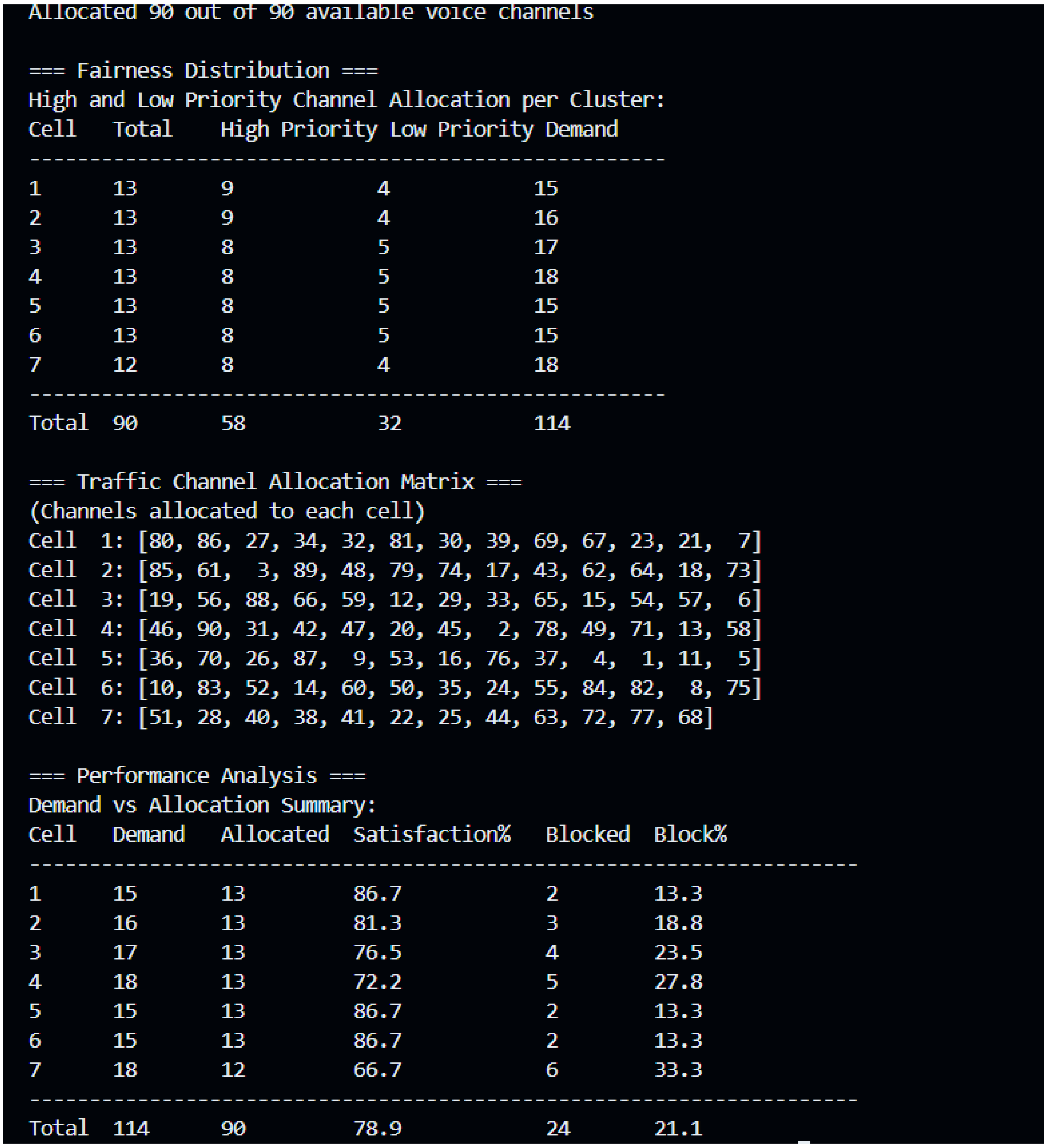


```
=== Control Channel Allocation Matrix ===
Total control channels: 10
Distribution across 7 cells:

Control Channel Assignment Matrix:
Cell  1: [ 1,  2]
Cell  2: [ 3,  4]
Cell  3: [ 5,  6]
Cell  4: [ 7]
Cell  5: [ 8]
Cell  6: [ 9]
Cell  7: [10]

=== Voice Channel Allocation ===

Channel Priority Assignment:
Low Priority Channels (Priority 1-2): 32 channels
Low Priority: 1 4 5 6 7 8 11 13 15 18 21 23 37 49 54 55 57 58 62 63 64 65 67 68 71 72 73 75 77 78 82 84
High Priority Channels (Priority 3-5): 58 channels
High Priority: 2 3 9 10 12 14 16 17 19 20 22 24 25 26 27 28 29 30 31 32 33 34 35 36 38 39 40 41 42 43 44 45 46 47 48 50 51 52 53 56 59 60 61 66 69 70 74 76 79 80 81 83 85 8
 88 89 90

=== Fairness Distribution Process ===
Using priority-based round-robin allocation
Allocated 90 out of 90 available voice channels

=== Fairness Distribution ===
High and Low Priority Channel Allocation per Cluster:
Cell   Total    High Priority Low Priority Demand
-----------------------------------------------------
1      13       9           4            15
2      13       9           4            16
3      13       8           5            17
4      13       8           5            18
5      13       8           5            15
6      13       8           5            15
7      12       8           4            18
-----------------------------------------------------
Total  90       58          32           114
```

Update Available

A new version of Autodesk Desktop Connector available. To upgrade, do NOT uninstall your current version if you are on 16.x. Instead, exit Desktop Connector and run the installer to upgrade.

Learn More



```
Allocated 90 out of 90 available voice channels

=== Fairness Distribution ===
High and Low Priority Channel Allocation per Cluster:
Cell    Total    High Priority Low Priority Demand
-----------------------------------------------------
1       13       9             4            15
2       13       9             4            16
3       13       8             5            17
4       13       8             5            18
5       13       8             5            15
6       13       8             5            15
7       12       8             4            18
-----------------------------------------------------
Total   90       58            32           114

=== Traffic Channel Allocation Matrix ===
(Channels allocated to each cell)
Cell  1: [80, 86, 27, 34, 32, 81, 30, 39, 69, 67, 23, 21,  7]
Cell  2: [85, 61,  3, 89, 48, 79, 74, 17, 43, 62, 64, 18, 73]
Cell  3: [19, 56, 88, 66, 59, 12, 29, 33, 65, 15, 54, 57,  6]
Cell  4: [46, 90, 31, 42, 47, 20, 45,  2, 78, 49, 71, 13, 58]
Cell  5: [36, 70, 26, 87,  9, 53, 16, 76, 37,  4,  1, 11,  5]
Cell  6: [10, 83, 52, 14, 60, 50, 35, 24, 55, 84, 82,  8, 75]
Cell  7: [51, 28, 40, 38, 41, 22, 25, 44, 63, 72, 77, 68]

=== Performance Analysis ===
Demand vs Allocation Summary:
Cell    Demand   Allocated  Satisfaction%   Blocked  Block%
-------------------------------------------------------------------
1       15       13         86.7            2        13.3
2       16       13         81.3            3        18.8
3       17       13         76.5            4        23.5
4       18       13         72.2            5        27.8
5       15       13         86.7            2        13.3
6       15       13         86.7            2        13.3
7       18       12         66.7            6        33.3
-------------------------------------------------------------------
Total   114      90         78.9            24       21.1
```

Conclusion:

The fundamental principle of dynamic channel allocation (DCA) is to manage the limited radio frequency spectrum more efficiently by assigning channels to cells on demand, rather than pre-assigning a fixed set of channels to each cell. In contrast to fixed channel allocation (FCA), where a cell is permanently assigned a specific group of channels, DCA allows all available channels in a system to be shared by all cells. An allocation algorithm is used to assign an available channel to a cell requesting it, as long as the new allocation does not interfere with existing calls. This flexibility is particularly useful in environments with fluctuating and unpredictable traffic, as it can significantly reduce the probability of call blocking.

The results of the simulation provide several key inferences about the effectiveness of dynamic channel allocation:

- **Blocking is a function of demand:** The experiment clearly demonstrates that when the total traffic demand from all cells exceeds the number of available voice channels, channel blocking will occur. This is evidenced by the "Blocked" and "Block %" columns in the performance analysis summary, which show that some calls could not be satisfied.
- **The power of allocation strategies:** The simulation highlights the importance of a well-designed allocation strategy. The priority-based, round-robin allocation method used in the experiment shows an attempt at fairness. While high-demand cells might have a lower satisfaction percentage due to competition for channels, the system as a whole still attempts to satisfy as many calls as possible by distributing the available channels. The use of both high and low-priority channels also shows how a system can be designed to favor certain types of traffic or users.
- **Maximizing spectrum utilization:** The simulation proves that the dynamic approach successfully allocates all 90 available voice channels, even when the total demand is higher than the supply, thereby maximizing spectrum utilization.

A great real-life example for an electronics undergraduate to understand dynamic channel allocation is the channel management in a modern Wi-Fi router. Imagine you live in an apartment building with many other Wi-Fi networks. Each network operates on a specific channel (usually from 1 to 11). If everyone's router is set to a fixed channel, say Channel 6, there will be massive interference and poor performance for everyone. However, modern routers use a form of DCA. When a router is powered on, it scans the environment to find the least-congested channel and dynamically selects it. If the traffic on that channel increases or interference is detected from a new source, the router might automatically switch to a different, less-used channel to ensure optimal performance. This on-the-fly decision-making is a perfect analogy for how dynamic channel allocation works in a cellular network, where the "router" is the base station and the "channels" are the frequency bands.