어드벤처디자인

(레벨 2)

2021년

김영학, 황준하, 김성영, 윤현주

이 자료는 금오공과대학교 컴퓨터공학과 2학년의 “어드벤처디자인” 과목에서 사용하기 위해 제작되었습니다.

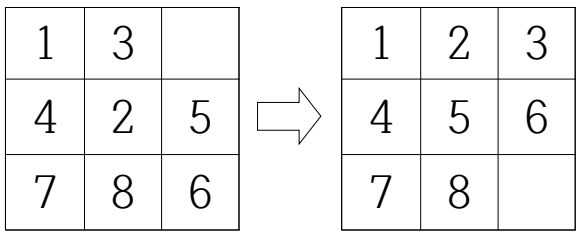
차 례

2. 레벨 2

2.1 8-Puzzle 게임

게임 개요

3x3 타일 위에 1부터 8까지의 숫자들이 놓여 있다. 초기 상태는 왼쪽 그림과 같이 헝클어진 상태이지만 공백 타일을 사용하여 숫자들을 옮김으로써 오른쪽과 같은 최종 상태로 만들 수 있다.



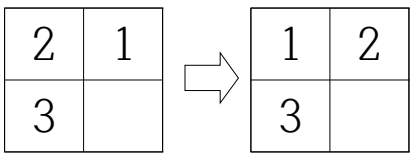
초기 상태 최종 상태

기본 문제 - 초기 상태 생성

8-Puzzle(3x3) 게임을 비롯하여 15-Puzzle(4x4), 24-Puzzle(5x5), ... 게임을 할 수 있는 게임 프로그램을 작성하라.

제한 요소 및 요구 사항

- 메뉴 방식으로 프로그램이 실행된다.
- 2개 이상의 소스파일과 1개 이상의 헤더파일을 사용한다.
- 초기 상태 생성 문제 해결 : 9개의 위치에 1부터 8까지의 숫자를 무작위로 위치시키면 풀 수 있는 퍼즐의 초기 상태가 만들어질까? 다음과 같은 3-puzzle로부터 해답을 구해보고, 무작위가 아닌 어떤 방법으로 초기 상태를 생성해야 하는지 고안하라.



- 전역 변수 사용은 가급적 자제한다.
- 키보드(또는 마우스)를 통해 다음 상태로의 이동이 가능하다.
- 현재 상태를 알기 쉽게 표현할 수 있다.
- 게임이 끝났는지(최종 상태와 같은지) 알 수 있다.
- 게임판의 크기를 조정할 수 있다.

확장 문제

① 현재 상태를 파일로 저장한 후 나중에 이 상태로부터 다시 시작할 수 있다.

② GUI 방식으로 구현한다.

③ 2인용 모드 : 두 명이 동시에 게임을 진행할 수 있다. (동일한 상태로부터 출발하여 누가 빨리 끝낼 수 있는지 대결)

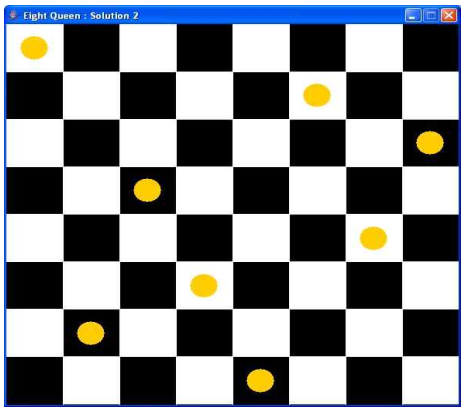
④ 컴퓨터가 스스로 이동하여 최종 상태를 만들 수 있다.

⑤ 컴퓨터가 최단 경로를 계산하여 스스로 게임을 실행하거나, 사용자 요구 시 힌트를 보여 주는 기능을 추가한다. (본 기능이 구현되는 경우 상대평가 시 반영)

2.2 8-Queens Problem

개요

8-Queens 문제는 8 × 8 체스보드에서 8개의 여왕이 서로를 공격할 수 없도록 배치하는 문제이다. 체스에서 여왕은 장애물이 없을 경우 전후좌우 그리고 대각선까지 원하는 칸 수만큼 이동할 수 있다. 즉 체스보드에서 8개의 여왕을 어떻게 배치하면 서로 공격할 수 없는 위치에 놓을 수 있는가에 대한 문제이다. <그림 2.2.1>은 8개의 여왕 문제에 대한 여러 가지 해 중의 하나를 보여준다.



<그림 2.2.1>

프로그램 입출력

○ 입력 : 체스보드의 크기를 입력 (4~N(?) 사이의 수) - 체스보드의 크기는 사용자가 입력하며 4×4 ~ N×N(?) 크기 내에서 입력하도록 함

○ 출력 : 다음과 같은 양식으로 출력(4개의 여왕 문제 예)

Row 1 - Col 2		Q		
Row 2 - Col 4				Q
Row 3 - Col 1	Q			
Row 4 - Col 3			Q	

문제 해결 방법 예시

이 문제는 스택(stack)과 백트래킹(backtracking)을 사용하여 해결할 수 있다. 4×4 크기를 갖는 체스보드에서 해결 과정을 분석해 보자(<그림 2.2.2> 참조).

▷ 체스보드 (1,1)의 위치에 첫 번째 여왕을 위치시키고 이 위치를 스택에 삽입(push)한다. - step1

▷ 한 행에 하나의 여왕만 배치할 수 있으므로, 두 번째 행에서 여왕을 배치할 수 있는 위치를 찾는다. (2,1)과 (2,2)는 (1,1)에 위치한 여왕과 직선 및 대각선에 위치하므로 가능하지 않다. 따라서 (2,3)의 위치에 두 번째 여왕을 놓고 그 위

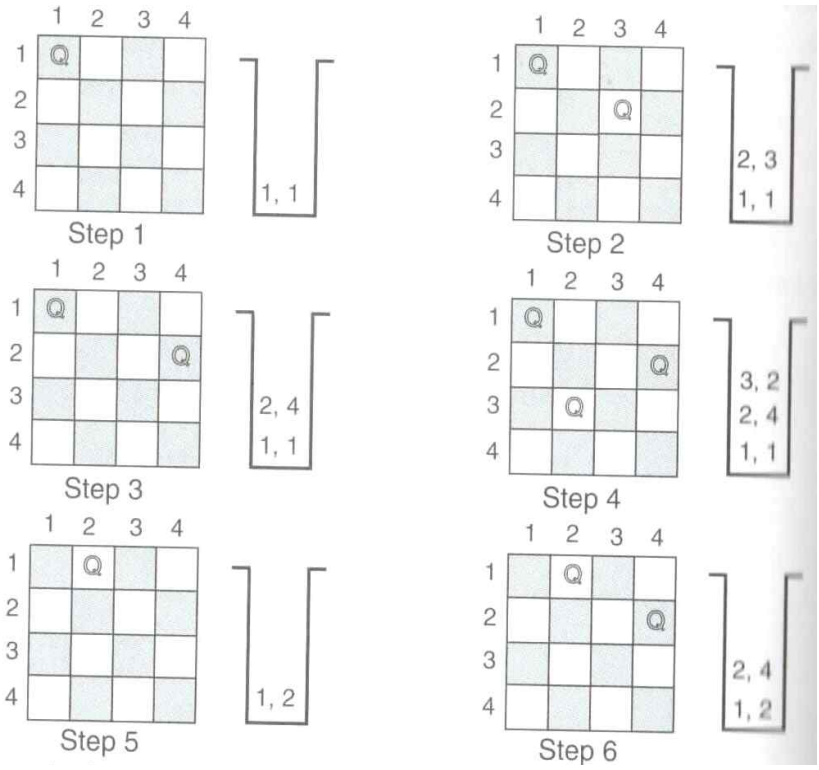
치의 인덱스를 스택에 삽입한다. - step 2

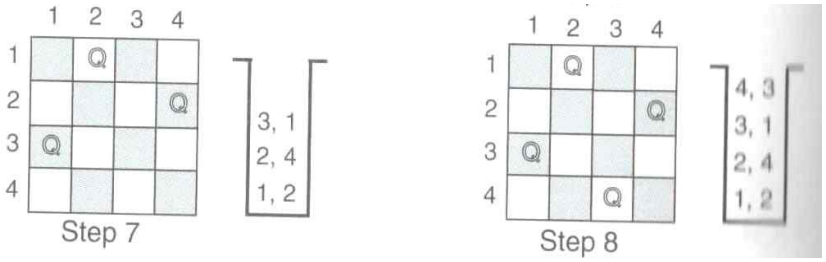
▷ 세 번째 행에서 여왕의 위치를 찾는다. 그런데 세 번째 행에서는 어떤 위치도 가능하지 않다. 따라서 스택에서 항목을 삭제(pop)하고 두 번째 행으로 백트래킹한 후 두 번째 행의 여왕에 대한 새로운 위치를 찾는다. 두 번째 행에서 (2,4) 위치가 가능하기 때문에 이 인덱스 값에 두 번째 여왕을 놓고 스택에 삽입한다. - step 3

▷ 세 번째 행을 다시 살펴본다. 이제 (3,2)의 위치에 세 번째 여왕을 배치한다. - step 4

▷ 네 번째 행에 놓일 여왕의 위치를 찾는다. 그러나 네 번째 행의 어떤 열에도 여왕을 놓을 수 없다. 이전 세 행에 이미 놓인 여왕들과 같은 행이나 열 또는 대각선상에 있기 때문이다. 이러한 상황을 해결하기 위해 스택에 있는 자료의 삭제를 통하여 백트래킹을 하면서 각 행에서 가능 여부를 조사한다. 예에서는 모든 행이 가능하지 않기 때문에 첫 번째 행에서의 위치를 (1,2)로 변경해야 한다. - step 5

▷ 이제 (1,2)에 놓인 첫 번째 여왕을 고려하여 두 번째 행의 (2,4), 세 번째 행의 (3,1)에 각각의 여왕을 놓고 마지막으로 네 번째 행의 (4,3) 위치에 마지막 여왕을 놓는다. 그림의 step 6, step 7, step 8을 차례로 참고하라.





<그림 2.2.2>

기본 문제

주어진 체스판의 크기에 대한 여왕 문제의 해를 구하시오. 해는 여러 개가 존재할 수 있으므로 모든 해를 구하되 각 크기에 대해 몇 개나 존재하는지 확인한다.

제한요소 및 요구사항

- 기본 메뉴를 제공한다.
- 2개 이상의 소스파일과 1개 이상의 헤더파일을 사용한다.
- 전역 변수 사용은 가급적 자제한다.
- 모든 해를 구하고 출력한다. (주어진 보드 크기에 대해 몇 개의 해가 존재하는지를 확인한다)

확장 문제

① GUI 방식으로 구현한다.
② 해를 탐색하는 과정을 차례로 보여준다(Step by Step).
③ N값이 큰 경우, 모든 해를 구하는데 과도한 시간이 걸릴 수 있다. N 값이 작을 때의 해의 개수 및 실행 시간으로 N값이 큰 경우에 대한 해의 개수 및 실행 시간을 유추해 보라.
④ 멈춤 기능 제공

2.3 압축된 BMP 이미지 파일에 대한 영상 처리

BMP 파일의 구조 및 특징

① BMP 파일 구조

BMP 파일은 그림 1과 같이 헤더와 이미지 데이터로 이루어진다.

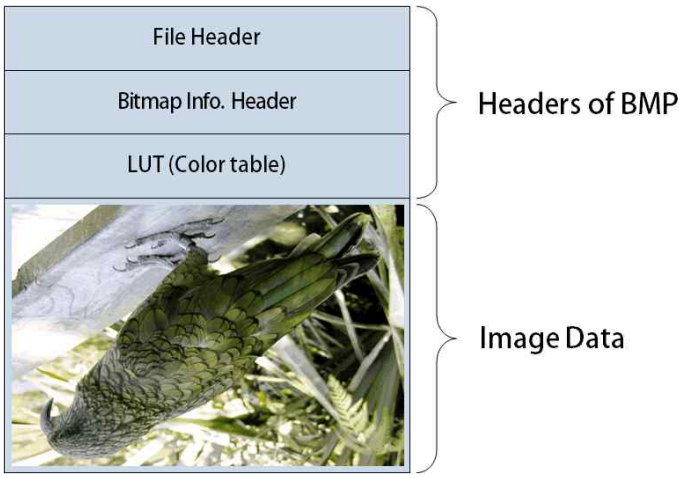


그림 1. BMP 파일 구조

헤더는 파일 헤더, 비트맵 정보 헤더와 칼라 테이블 영역으로 구성된다. 각 영역의 구성은 다음과 같다.

```
/* 자료형 재선언 */
typedef unsigned char       BYTE;
typedef unsigned short      WORD;
typedef unsigned long       DWORD;

/* 파일 헤더의 멤버 */
WORD  bfType;           // BMP file type: "BM" (4D42)
DWORD bfSize;           // 전체 파일의 크기 (byte)
WORD  bfReserved1;      // reserved (항상 0)
WORD  bfReserved2;      // reserved (항상 0)
DWORD bfOffBits;        // 이미지 데이터의 시작 오프셋
```

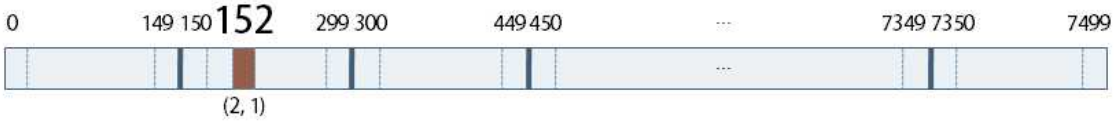
```
/* 비트맵 정보 헤더의 멤버 */
DWORD biSize;            // 구조체의 크기 (bytes)
LONG  biWidth;           // 비트맵의 가로 길이 (pixels)
LONG  biHeight;          // 비트맵의 세로 길이 (pixels)
WORD  biPlanes;          // 비트 플레인 수 (항상 1)
WORD  biBitCount;        // 픽셀당 비트수 (1,4,8,16,24,32)
DWORD biCompression;     // 압축 유형 (BI_RGB, BI_RLE4, BI_RLE8)
DWORD biSizeImage;       // 비트맵 데이터의 크기 (bytes)
LONG  biXPelsPerMeter;   // 수평 해상도 (pixels/meter)
LONG  biYPelsPerMeter;   // 수직 해상도 (pixels/meter)
DWORD biClrUsed;         // LUT(Lookup Table)에 포함된 칼라 인덱스의 개수
DWORD biClrImportant;    // 비트맵을 화면에 출력하기 위해 사용된 칼라
                            인덱스의 개수

/* 칼라 테이블 */
BYTE  rgbBlue;           // 파란색 성분 (B component)
BYTE  rgbGreen;          // 녹색 성분 (G component)
BYTE  rgbRed;            // 빨간색 성분 (R component)
BYTE  rgbReserved1;      // 예약 (reserved)
```

windows.h 파일에는 BMP 파일 처리를 위한 자료형 재선언과 BMP 헤더를 나타내는 구조체가 이미 마련되어 있다. 파일 헤더를 위한 BITMAPFILEHEADER, 비트맵 정보 헤더를 위한 BITMAPINFOHEADER, 칼라 테이블을 위한 RGBQUAD이다. 단 RGBQUAD는 하나의 색을 저장하기 위한 자료구조이다.

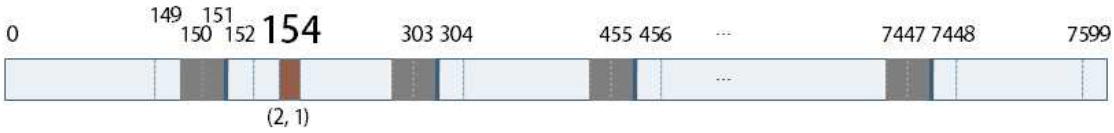
칼라 테이블에서는 하나의 색을 4개의 바이트(red, green, blue, reserved)를 사용해 나타내며, 비트맵 정보 헤더의 biClrUsed 멤버의 개수만큼 색이 포함된다(256색 이하인 경우 사용). 단, gray-scale 영상은 이 필드의 값이 0이지만(모든 색상을 다 사용한다는 의미) 칼라 테이블은 존재하며 색의 개수는 $2^{biBitCount}$이다. 이때 biBitCount는 비트맵정보 헤더의 멤버이며 픽셀 당 비트수를 나타낸다. 8bit gray-scale 영상에서는 칼라 테이블에 256개(2^8)의 색을 포함한다.

② BMP 파일 구조의 특징

첫째, 영상 데이터의 각 행은 4byte의 배수로 구성된다. 예를 들어, 가로 150개 픽셀, 세로 100개 픽셀로 구성되는 150×100 크기의 영상을 생각해 보자. 만약 8bit gray-scale 영상인 경우 픽셀 당 8bit를 가지므로 한 행의 바이트 수는 150byte이지만 150은 4의 배수가 아니므로 한 행은 4의 배수인 152byte가 되고, 따라서 영상 데이터에는 포함되지 않는 2byte가 추가로 포함(패딩)된다(그림 2 참조). 그림 2에서 패딩이 없는 경우에는 (x, y)=(2, 1)인 위치의 픽셀에 대한 접근을 위한 인덱스는 152(1*150+2)이지만, 패딩이 있는 경우는 154(1*152+2)이다.



(a) 패딩이 없는 경우



(b) 패딩이 있는 경우

그림 2. BMP 한 행의 구성

두 번째 특징은 영상 데이터가 상하 반전되어 파일에 저장된다는 것이다. 따라서 마지막 행이 가장 먼저 저장되고 그 다음으로 마지막 바로 위의 행이 저장되며 첫 번째 행은 가장 마지막에 저장된다(그림 3 참조).

그림 3. 영상 데이터의 상하 반전

영상 데이터의 (x, y) 좌표에 위치하는 픽셀에 접근하기 위해서는 다음의 수식을 사용할 수 있다. 이 때 widthStep은 한 행 당 바이트 수(4바이트의 배수)이다. 예를 들어, height가 10이고 widthStep이 152일 때 (x, y)가 (0, 9), (0, 8)일 때 픽셀의 위치는 각각 0과 152가 된다.

$$(x, y) \Rightarrow [widthStep^{*}(height\text{-}y\text{-}1) + x]$$

BMP RLE 복원

BMP는 픽셀 당 비트 수가 1, 4, 8인 경우 영상 데이터를 RLE 방식으로 압축하여 저장할 수 있다. RLE는 단순한 무손실 데이터 압축 방법으로 동일한 값이 연속(run)으로 나타나는 경우 그 값과 발생 횟수(length)를 저장하여 데이터양을 줄이는 압축 방법이다. 예를 들어, 데이터 열이 “AAAAABB”로 구성된 경우 A가 다섯 번, B가 두 번 나타나므로 5A2B로 표현할 수 있다. 7바이트의 정보를 4바이트로 줄일 수 있다.

BMP의 RLE는 두 가지 모드(encoded, absolute)에서 압축을 수행하며 특별한 의미의 마커들을 포함한다. **Encoded mode**는 일반적인 RLE 부호화/복호화 과정을 의미한다. 이 모드에서는 두 개의 바이트를 처리 단위로 사용하여 첫 번째 바이트는 발생 횟수, 두 번째 바이트는 값을 나타낸다. 다음 예를 확인하자. 부호화 코드인 02 00은 복원하면 00의 값이 2번 발생하므로 00 00으로 해석할 수 있다. 동일하게 부호화 코드인 05 0A는 0A의 값이 5번 발생하여 0A 0A 0A 0A 0A로 복원할 수 있다.

부호화 코드(encoded code)	복호화 코드(decoded code)
02 00 05 0A	00 00 0A 0A 0A 0A 0A

만약 횟수를 나타내는 첫 번째 바이트가 0(00)이고 두 번째 바이트가 3이상(<256) 값이면 **absolute mode**를 의미한다. 이 모드는 연속된 값이 등장하고 않고 값이 계속해서 변하는 경우에 사용하는데 이 모드에서 표현 가능한 최소 길이는 3, 최대 길이는 255이다. 첫 번째 값은 **absolute mode**를 구분하기 위한 표시자이고 두 번째 값은 데이터 열의 길이이다. 따라서 이 값만큼의 부호화되지 않은 원래의 데이터가 나타난다. 만약 데이터 열의 길이가 홀수이면 부호화 코드의 마지막에 0을 추가한다. 다음 예를 확인하자. 첫 번째 코드인 00 04 00 01 02 03은 데이터 열의 길이가 4(04)이고 실제 데이터는 00 01 02 03이다. 두 번째 코드에서는 데이터 열의 길이가 5이므로 홀수이다. 따라서 부호화 코드의 마지막에 0(00)을 추가로 포함하고 있다.

부호화 코드(encoded code)	복호화 코드(decoded code)
00 04 00 01 02 03 00 05 00 01 02 03 04 <u>00</u>	00 01 02 03 00 01 02 03 04

데이터 열의 길이가 1 혹은 2인 경우에는 absolute mode 대신 encoded mode를 사용한다. Encoded mode에서 발생 횟수를 1로 지정하여 부호화한다. 다음 예를 확인하자. 첫 번째 부호화 코드인 01 00은 00이 1번 나타나므로 00으로 해석할 수 있다. 두 번째 코드인 01 01 01 02는 01인 1번 02가 1번 나타나는 것으로 해석하여 01 02로 복원할 수 있다.

부호화 코드(encoded code)	복호화 코드(decoded code)
01 00 01 01 01 02	00 01 02

발생 횟수를 나타내는 첫 번째 바이트가 0이고 두 번째 바이트가 0, 1, 2 중의 한 가지 값이면 특별한 의미를 갖는 **마커(marker)**에 해당한다. 마커가 0이면 "라인의 마지막 위치(End of Line)", 1이면 "영상 데이터의 마지막 위치(End of

Bitmap)" 2이면 "델타(Delta)"를 의미한다.

■ End of Line (00 00)

이 마커는 다음 코드는 새로운 라인을 위한 것이며 현재 라인을 위한 더 이상은 정보는 없음을 의미한다. 만약 영상의 가로길이보다 픽셀의 개수가 적으면 ~~마지막 값으로 저장~~(0으로 저장?)한다.

■ End of Bitmap (00 01)

이 마커는 부호화된 데이터의 마지막을 나타낸다. 나머지 영역은 ~~마지막 값으로 저장~~(0으로 저장?)한다.

■ Delta (00 02)

이 마커는 현재 위치로부터 상대적으로 이동할 위치를 나타낸다. 즉 현재 위치에서 새로운 위치 사이에 있는 데이터(픽셀)는 부호화를 생략한다. Delta 마커 이후에 등장하는 첫 번째 바이트는 수평방향의 오프셋, 두 번째 바이트는 수직방향의 오프셋을 의미한다. 단, 이 값들은 부호 없는 정수로 처리한다. 오프셋의 다음에 나타나는 코드는 새로운 위치의 픽셀을 위한 것임을 알 수 있다. 생략한 픽셀들은 ~~현재 위치의 값으로~~(0으로 저장?) 채운다.

기본 문제

8bit gray-scale 영상의 BMP 파일을 읽어 **영상의 밝기를 반전하고 사용자가 지정한 크기의 사각형을 그린 뒤에 새로운 이름으로 저장**하는 프로그램을 작성하시오. 단, 입력으로 사용하는 BMP 파일은 run length encoding (RLE) 방식으로 **압축(BI_RLE8)**되어 있다.

제한요소 및 요구사항

- 기본 메뉴를 제공한다.
- 2개 이상의 소스파일과 1개 이상의 헤더파일을 사용한다.
- 전역 변수 사용은 가급적 자제한다.
- 압축하지 않은 형식과 RLE 압축한 형식을 모두 지원한다.

확장 문제

① 2가지 이상의 영상 처리 기능을 제공한다.
② 24bit true-color 영상을 함께 지원한다.
③ PPM 파일을 지원하고 결과 파일의 종류(저장 형식)를 변경할 수 있다. (Option : PGM 파일 지원)
④ GUI 환경을 제공한다.

2.4 태피 상점의 큐 시뮬레이션

개요

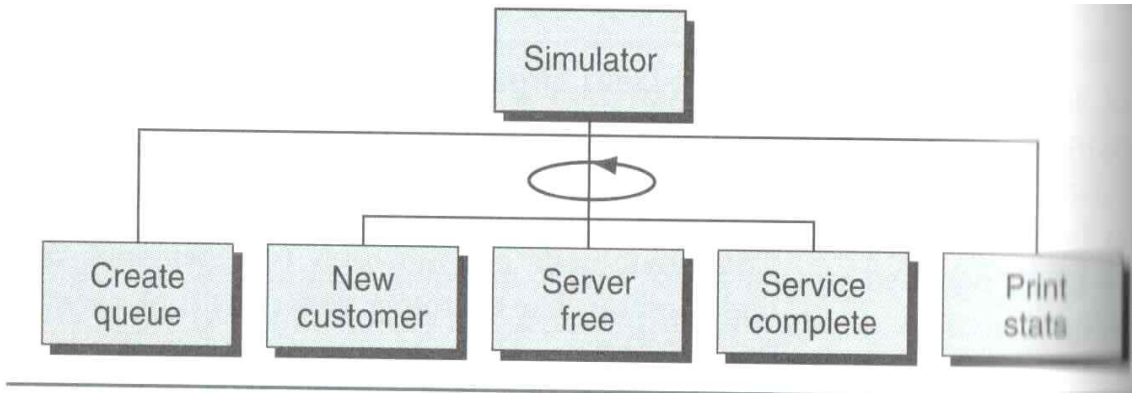
은행이나 학교 식당 등 순차적인 서비스가 일어나는 시스템을 모델링하기 위해 보통 큐(queue)를 사용한다. 그리고 이러한 큐를 이용해 실제 동작이 어떤 방식으로 일어날지 컴퓨터 프로그램으로 시뮬레이션할 수 있다. 시뮬레이션은 다음과 같이 진행된다.

1) 상점은 한 개의 창구만을 가지며 한 명의 점원이 한 번에 한 명의 고객만을 서비스 한다.
2) 고객을 서비스하는 시간은 1분에서 10분 사이가 소요된다.
3) 상점은 하루 8시간, 주 7일 업무를 한다. (상점의 하루의 일을 시뮬레이션하기 위해 480분(8시간 × 60분)을 갖는 큐 모델을 생성한다.)
4) 시뮬레이션은 이벤트가 1분 간격으로 시작하고 정지하는 디지털 클록을 사용한다. 즉, 고객들은 분단위로 도착하여 큐에서 분단위로 합산된 시간만큼 대기하고 임의의 시간만큼 서비스된다. 매 분단위의 시뮬레이션은 3개의 이벤트인 고객의 도착시간, 고객의 서비스 시작시간, 고객의 서비스 완료시간을 체크한다.
5) 최종적으로 이 시뮬레이션에서 주어진 시간 동안에 전체 고객의 수, 전체 서비스 시간, 평균 서비스 시간, 평균 대기시간 등의 통계 정보를 출력한다.

시뮬레이션을 위한 시스템 구성

□ 시뮬레이션 모듈 개요

아래의 그림에서 보인 것과 같이 태피 상점의 시뮬레이터는 5개의 모듈로 구성된다. “Create queue” 모듈은 시뮬레이션에 사용될 큐를 생성하고 초기화한다. “New customer” 모듈은 새로운 고객이 도착하여 고객의 정보가 큐에 삽입되어야 하는지를 결정한다. “Server free“ 모듈은 점원이 쉬고 있는 상태인지를 점검하며, 쉬고 있는 상태일 경우 큐에 고객이 존재하면 다음 고객을 서비스한다. “Service complete” 모듈은 현 고객이 완전하게 서비스 되었는지를 결정하고, 그렇다면 그 고객에 관한 통계 정보를 수집하여 출력한다. “Print stats“ 모듈은 큐 시뮬레이션에서 일어난 전체 통계 정보를 출력한다.



□ 이벤트

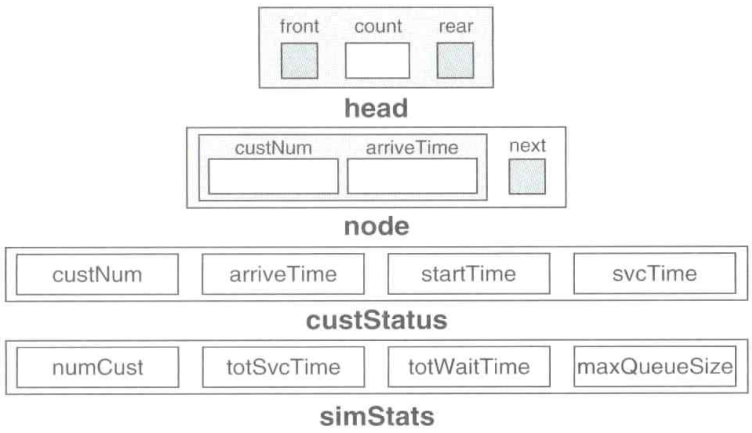
새로운 고객의 도착은 “New customer” 모듈에서 처리한다. 도착 비율을 결정하기 위해 그 상점은 며칠에 걸쳐 스톱워치를 사용하여 고객의 패턴을 조사하였다. 그 상점은 새로운 고객이 평균적으로 매 4분 간격으로 도착하는 것을 발견하였다. 따라서 시뮬레이션을 위해 1에서 4사이의 난수를 발생하여 도착 비율을 모의실험하며, 만일 난수 값이 4라면 새로운 고객이 도착하였다고 가정하고, 그렇지 않으면(난수 값이 1, 2, 3) 고객이 도착되지 않았다고 가정한다.

점원이 한가한 경우(idle) 새로운 고객의 서비스를 시작할 수 있다. “Server free” 모듈에서는 매 분단위로 점원이 서비스 중인지 쉬고 있는지를 결정한다. 점원이 쉬고 있을 경우 큐에서 다음 대기 고객이 서비스될 수 있으며, 그렇지 않으면 대기 고객은 큐에 그대로 유지된다.

최종적으로 매 분의 끝에 시뮬레이터는 현 고객에 대한 서비스가 완료되었는지를 결정한다. 현 고객에 대한 처리 시간은 발생된 난수에 의해서 결정된다. 시뮬레이션은 분 단위이므로 난수로 정해진 각 고객의 서비스 시간만큼 분단위 동작이 반복되어야 트랜잭션이 완료된다. 한 고객에 대한 서비스가 완전하게 완료되었을 때 판매에 관한 통계를 수집하고 점원은 한가한 상태로 전환된다.

□ 자료구조 설계

큐 시뮬레이션을 위해 4개의 자료구조가 사용된다. 4개의 자료구조는 큐 헤드, 큐 노드, 현 고객 상태, 시뮬레이션 통계이며 세부 내용은 다음 그림과 같다.



Queue Data Structures

○ 큐 헤드(head) : 큐에 대한 두 개 포인터인 front(삭제) 및 rear(삽입), 그리고 현재 큐에 대기 중인 고객 수를 가지는 count 변수를 포함한다.

○ 큐 노드(node) : 고객 번호와 큐에 도착한 시간과 다음 고객에 대한 포인터를 포함한다.

○ 고객 상태(custStatus) : 고객의 주문을 처리하는 동안 고객 상태정보를 유지하기 위해 고객번호, 도착시간, 서비스 시작시간, 서비스시간 등의 값을 저장한다.

○ 시뮬레이션 통계(simStats) : 시뮬레이션을 완료한 후에 전체 고객 수, 전체 서비스 시간, 평균 대기시간, 최대 큐의 크기 등에 관한 정보를 출력하기 위한 자료를 누적해 저장한다.

기본 문제

해변에 위치한 태피(taffy) 상점에서 일어나는 큐 모델을 시뮬레이션하는 프로그램을 개발하시오.

□ 입력

- 큐 시뮬레이션 시간(분 단위) : 최대 480분(1일 시간)
- 고객의 도착 : 1에서 4사이의 난수 값을 생성하여 4일 경우만 새로운 고객이 도착하는 것으로 가정
- 서비스 시간 : 1에서 10사이의 난수 값 발생 (1~10분)

□ 출력 : 다음의 시뮬레이션 통계를 출력

- 전체 통계 : 평균 큐 대기시간, 평균 서비스 시간, 전체 서비스 시간, 전체

고객 수
- 개별 고객 : 도착시간, 시작시간, 대기시간, 서비스시간

제한요소 및 요구사항

- 기본 메뉴를 제공한다.
- 2개 이상의 소스파일과 1개 이상의 헤더파일을 사용한다.
- 전역 변수 사용은 가급적 자제한다.
- 여러 번 실행하여 통계 자료를 제공한다.

확장 문제

① 서비스 창구의 개수를 증가하면서 처리 결과의 차이를 확인한다.
② 2개 이상의 서비스 창구가 존재하는 경우 다중 스레드를 사용한다.
③ 전화 서비스를 제공한다.
④ GUI 환경을 제공한다.

2.5 순회 외판원 문제

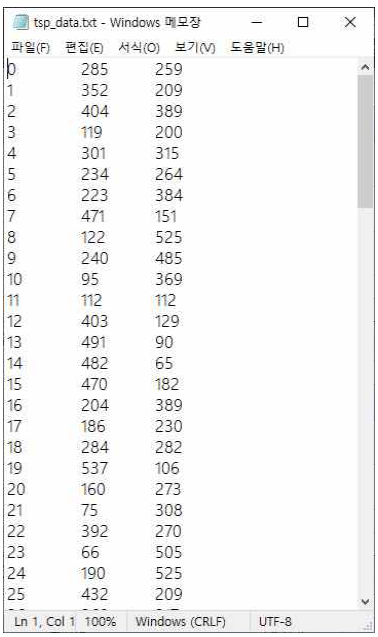
개요

순회 외판원 문제(TSP, Traveling Salesman Problem)는 여러 도시가 주어질 때 하나의 도시로부터 출발하여 모든 도시를 단 한 번씩만 경유하고 다시 출발 도시로 되돌아오되 최단 경로를 찾는 문제로 정의된다.

순회 외판원 문제를 대상으로 최단 경로 또는 최단 경로는 아니지만 주어진 시간 내에 가능하면 더 짧은 경로를 찾기 위한 프로그램을 만들어 보자.

기본 문제

다음과 같이 데이터 파일(tsp_data.txt)에는 총 100개의 도시 데이터가 주어져 있는데, 도시 번호와 x 좌표, y 좌표가 주어진다. 출발 도시는 0번 도시로 가정한다. 도시 사이의 거리는 직선 거리, 즉 유클리디언 거리로 계산하되 소수점 이하 값이 나오는 경우 소수점 첫 번째 자리에서 반올림하여 사용한다. 사용자로부터 몇 개의 도시에 대한 최단 거리를 구할 것인지 받아들이고(예를 들어, 5개 도시라고 하면 0번 도시부터 4번 도시까지의 총 5개 도시를 대상으로 한다) 모든 경로들 중 최단 경로를 구하고 그 결과로 방문 도시의 순서와 거리를 출력해 보라.



이와 같이 모든 경우의 수를 나열하여 최적의 답의 찾는 탐색 방법을 완전 탐색(Exhaustive Search)이라 한다.

기반 알고리즘 설계 : 순열 생성

5개의 도시가 있다고 가정하자(0, 1, 2, 3, 4번 도시를 의미함). 0번 도시가 출발 도시이므로 첫 번째 도시는 항상 0번 도시가 된다. 따라서 0번 도시를 제외하고

1, 2, 3, 4번 도시를 한 번씩 모두 방문하고 돌아오는 경로를 모두 열거한 후 0번 도시부터 해당 경로를 따라 다시 0번 도시로 돌아올 때의 거리를 합산하면 어떤 경로가 최적(최단) 경로인지 바로 알 수 있다. 즉, 다음과 같은 총 24개의 경로(4!)에 대한 방문 거리를 각각 구해 비교하면 되는 것이다.

1, 2, 3, 4
1, 2, 4, 3
1, 3, 2, 4
1, 3, 4, 2
1, 4, 2, 3
1, 4, 3, 2
2, 1, 3, 4
2, 1, 4, 3
2, 3, 1, 4
2, 3, 4, 1
2, 4, 1, 3
2, 4, 3, 1
3, 1, 2, 4
3, 1, 4, 2
3, 2, 1, 4
3, 2, 4, 1
3, 4, 1, 2
3, 4, 2, 1
4, 1, 2, 3
4, 1, 3, 2
4, 2, 1, 3
4, 2, 3, 1
4, 3, 1, 2
4, 3, 2, 1

따라서 먼저 본 문제를 풀기 위해서는 모든 경로를 나열하는 순열 생성 알고리즘이 필요하므로 순열 생성 알고리즘을 설계해야 한다.

다양한 순열 생성 알고리즘이 존재한다. 반드시 본인이 이해하고 직접 구현이 가능한 알고리즘을 선택하도록 한다.

순열 생성 알고리즘의 예 : 재귀 함수 및 Swap 함수 활용

data가 { 1, 2, 3 }와 같이 존재할 때 { 1, 2, 3 }, { 1, 3, 2 }, { 2, 1, 2 }, ... { 3, 2, 1 }과 같은 모든 순열을 생성한다.

함수 프로토타입은 다음과 같다.

void Permutation(ary, start_index, end_index);

최초 함수 호출은 다음과 같다.

Permutation(data, 0, 2);

Permutation 함수 내부는 다음과 같이 동작한다.

1) start_index와 end_index가 같으면 하나의 순열이 완성되었다는 의미이므로 현재 ary의 값들을 출력하고 함수를 빠져나간다. 물론 순회 외판원 문제 측면에서는 현재 경로를 대상으로 이동 거리를 계산하고 지금까지의 가장 좋은 경로보다 좋다면 가장 좋은 경로를 현재 경로로 업데이트한다.
2) 1)이 아니라면, start_index부터 end_index까지 각각의 값에 대해
   2-1) Swap(ary[start_index], ary[i]) : start_index에 있는 데이터 값과 i번째에 있는 데이터 값을 교환한다.
   2-2) Permutation(ary, start_index + 1, end_index) 함수를 재귀 호출한다.
   2-3) Swap(ary[start_index], ary[i]) : 교환한 값을 다시 교환한다.

예를 들어, Permuation({1, 2, 3}, 0, 2)에서 start_index(0)와 i(0)의 값이 교환되면 그대로이며 이제 Permutation({1, 2, 3}, 1, 2)이 실행된다. 그러면, start_index(1)과 i(1)의 값이 교환되고(그대로임), Permuation({1, 2, 3}, 2, 2)가 실행된다. 이때 start_index와 end_index의 값이 동일하므로 {1, 2, 3}이 하나의 완성된 순열이 된다.

이제 다시 start_index(1)의 값과 i(1)의 값을 서로 교환하고(그대로임), i값은 2가 되며, start_index(1)와 i(2)의 값을 서로 교환한다. 그러면 Permuation({1, 3, 2}, 2, 2)가 호출되고 start_index와 end_index의 값이 동일하므로 {1, 3, 2}가 다음 하나의 완성된 순열이 된다.

함수로부터 반환되면 start_index(1)와 end_index(2) 사이의 모든 값에 대해 처리가 완료되었으므로 Permutation 함수를 빠져나간다.

이전 Permutation 함수에서는 i가 0에서 1이 되고, Swap(ary[0], ary[1])과 Permutation({2, 1, 3}, 1, 2)가 실행된다.

이상과 같이 Swap, Permuation, Swap 함수가 반복적으로 실행되어 {2, 1, 3}, {2, 3, 1}, {3, 2, 1}, {3, 1, 2}의 순열이 계속해서 생성된다.

제한 요소 및 요구 사항

- 메뉴 방식으로 프로그램이 실행된다. 도시 개수 선택 등
- 2개 이상의 소스파일과 1개 이상의 헤더파일을 사용한다.
- 첫 번째 경로와 거리를 출력하고 더 좋은 경로가 나오면 그때의 경로와 거리를 출력한다. 최종적으로 가장 좋은 경로 및 거리를 출력한다.
- 전역 변수 사용은 가급적 자제한다.
- 실험 당 최대 수행 시간을 30분 이내로 제한하고, 30분 이내에 몇 개의 도시까

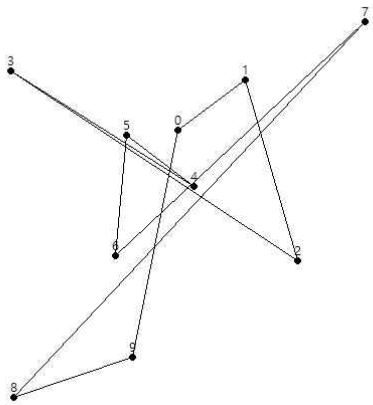
지 최적의 경로(모든 경로를 다 검사)를 찾을 수 있는지 확인한다. 대략적으로 5개, 10개, 11개, 12개, 13개, ...와 같이 실험을 실행해 보면 된다.

확장 문제

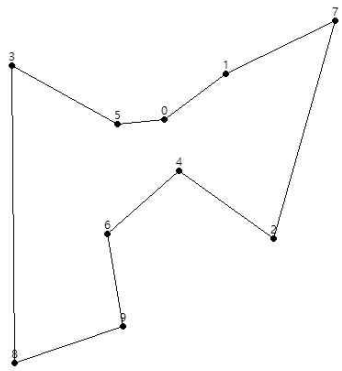
① 언덕 오르기 탐색 알고리즘을 적용하여 5개(0~4), 9개, 10개, 11개, 12개, 13개, 14개, 15개, 16개, 20개, 30개, 50개, 100개의 도시에 대해 적용해 보라. 각 실험 당 최대 수행 시간은 3분으로 제한한다. 실험 결과를 완전 탐색과 비교해 보라. 언덕 오르기 탐색 알고리즘은 다음과 같다.

```
1: Algorithm 언덕_오르기_탐색
2: 현재해 ← 초기해 생성 (예, 1~n을 무작위로 섞기)
3: while 경과 시간이 180초 이내 do
5:       다음해 ← 이웃해 생성 (예, 임의의 2개 위치를 교환)
6:       if 거리(다음해) ≤ 거리(현재해) then
7:            현재해 ← 다음해
8: return current
```

② GUI 방식으로 구현한다. 예시(10개 도시 : 0~9) : 초기 경로(및 거리)를 보여주고 중간에 더 좋은 경로가 나타나면 업데이트한다. 따라서 최종적으로 가장 좋은 경로가 나타난다.



[초기 경로의 예]



[최종 경로의 예]

③ 멈춤 기능 제공

④ 언덕 오르기 탐색 알고리즘에서 이웃해를 생성하는 방법을 다르게 적용해 보라. 예) 두 개의 위치를 선택하고 사이에 있는 도시 방문 순서를 역전(inversion)시킴

7 4 1 5 2 8 6 3

7 4 6 8 2 5 1 3