**알고리즘**

**설계 프로젝트**

|  |  |
| --- | --- |
| 프로젝트 명 | *Sudoku with Dancing Link Algorithm* |
| 팀 명 | *Let's Think* |
| 문서 제목 | 중간보고서 |

|  |  |
| --- | --- |
| **Version** | 1.0 |
| **Date** | 2014-NOV-24 |

|  |  |
| --- | --- |
| **팀원** | 유 동창 |
| 김 경동 |
| 김 성중 |
| 김 태욱 |
| 김 환규 |
| 박 훈 |
| 신 혜수 |
| **지도교수** | 최 준수 교수 |

|  |
| --- |
| **CONFIDENTIALITY/SECURITY WARNING**  이 문서에 포함되어 있는 정보는 국민대학교 전자정보통신대학 컴퓨터공학부 및 컴퓨터공학부 개설 교과목 알고리즘 수강 학생 중 프로젝트 “Sudoku with Dancing Link Algorithm”를 수행하는 팀 “Let’s Think”의 팀원들의 자산입니다. 국민대학교 컴퓨터공학부 및 팀 “Let’s Think”의 팀원들의 서면 허락없이 사용되거나, 재가공 될 수 없습니다. |

**문서 정보 / 수정 내역**

|  |  |
| --- | --- |
| **Filename** | 중간보고서- Let's Think -Sudoku.doc |
| **원안작성자** | 유동창, 김태욱, 김경동, 김성중, 박훈, 신혜수 |
| **수정작업자** | 김환규 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 수정날짜 | 대표수정자 | Revision | 추가/수정 항목 | 내 용 |
| 2014-11-24 | 김환규 | 1.0 | 최초 작성 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**목 차**

[**1** **프로젝트 목표** 4](#_Toc404678238)

[**2** **수행 내용 및 결과** 5](#_Toc404678239)

[**2.1** **개념 설명** 5](#_Toc404678240)

[**2.1.1** **Backtracking** 5](#_Toc404678241)

[**2.1.2** **Dancing Link with algorithm X** 5](#_Toc404678242)

[**2.2** **수행 내용** 9](#_Toc404678243)

[**2.2.1** **Backtracking** 9](#_Toc404678244)

[**2.2.2** **Dancing Link** 13](#_Toc404678245)

[**3** **수정된 연구내용 및 추진 방향** 16](#_Toc404678246)

[**3.1** **수정사항** 16](#_Toc404678247)

[**4** **부록** 17](#_Toc404678248)

[**4.1** **Backtracking** 17](#_Toc404678249)

[**4.2** **Dancing Link** 20](#_Toc404678250)

# **프로젝트 목표**

**Sudoku 문제와 문제를 해결한다.**

**수도쿠(Sudoku)는 9X9 크기의 행렬에 미리 몇 개의 원소에 한 자리 숫자가 주어져 있을 때, 모든 행과 모든 열에 1 부터 9 까지의 한 자리의 수가 각각 한 번만 나타나며, 또한 모두 9개의 3X3 크기의 작은 행렬에 있는 9 개의 원소에도 1 부터 9 까지의 한자리의 수가 각각 한 번만 나타나도록 빈 칸에 있는 행렬의 원소를 정하는 것이다.**

**이러한 수도쿠 문제를 해결하기 위해 Backtracking의 방법을 사용해서 우선 해결을 한다. 하지만 Backtracking의 방법은 답을 구할 때까지 가능성이 있는 경우에 대해서 모두 탐색하기 때문에 시간이 오래 걸린다.**

**따라서 Dancing Link Algorithm을 적용하여 Backtracking의 방법으로 구현했을 때보다 더 큰 크기의 행렬을 대상으로 수행하는 경우에도 잘 돌아갈 수 있도록 한다.**

# **수행 내용 및 결과**

## **개념 설명**

### **Backtracking**

루트 노드에서 리프 노드(leaf node)까지의 경로는 해답후보(candidate solution)가 되는데, 깊이우선검색을 하여 그 해답후보 중에서 해답을 찾을 수 있다. 그러나 이 방법을 사용하면 해답이 될 가능성이 전혀 없는 노드의 후손노드(descendant)들도 모두 검색해야 하므로 비효율적이다.

그래서 노드의 유망성을 확인한다. 전혀 해답이 나올 가능성이 없는 노드는 유망하지 않다(non-promising)고 하고(해당 노드의 서브트리 검색은 무의미), 그렇지 않으면 유망하다(promising)고 한다. 어떤 노드의 유망성을 점검한 후, 유망하지 않다고 판정이 되면 그 노드의 부모노드(parent)로 돌아간다.(“Backtrack”). 그런 다음, 다시 후손노드에 대한 검색을 계속하게 되는 절차이다.

Backtracking 알고리즘은 상태공간트리에서 깊이우선탐색을 실시하는데, 유망하지 않은 노드들은 가지쳐서(pruning) 검색을 하지 않으며, 유망한 노드에 대해서만 그 노드의 자식노드(children)를 검색한다.

이 알고리즘은 다음과 같은 절차로 진행된다.

1. 상태공간트리의 깊이우선탐색을 실시한다.

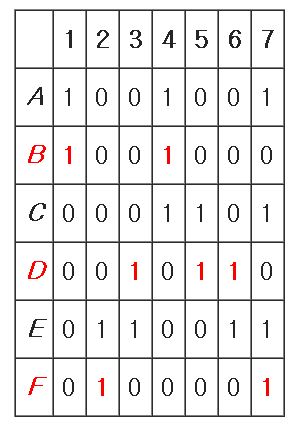
2. 각 노드가 유망한지를 점검한다.

3. 만일 그 노드가 유망하지 않으면, 그 노드의 부모노드로 돌아가서 검색을 계속한다.

### **Dancing Link with algorithm X**

**- Exact Cover Problem**

“Algorithm X”는 Exact Cover Problem을 쉽게 해결할 수 있는 알고리즘이다. 이때 “Algorithm X”에 사용되는 특별한 자료구조가 “Dancing Link”이다. Exact Cover문제는 0 또는 1로 이루어진 매트릭스에서 특정 행(row)들을 뽑아 모든 열이 1이 되게 하는 문제이다. 예를 들면 아래 그림에서 모든 열이 1이 되게 하는 행(row)들의 조합은 {B, D, F}이다. 따라서 <그림 1>에 대한 Exact Cover Problem의 해답(solution)은 {B, D, F}라고 할 수 있다. Exact Cover Problem은 특정 조건을 만족하는 해의 집합을 골라내는 문제에 적용될 수 있다.



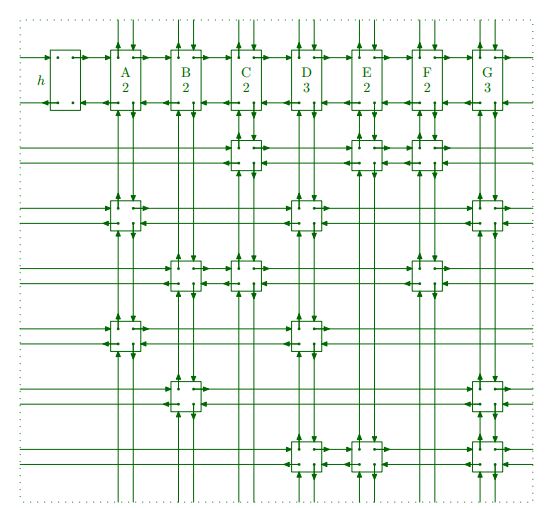
<그림 1>

**- Algorithm X**

Algorithm X는 Exact Cover Problem을 해결하기 위해서 특별한 매트릭스를 이용한다. Algorithm X에서 필요로 하는 매트릭스의 각 행(row)는 candidate 해를 의미하고, 각 열(column)은 candidate 해가 지켜야 할 조건(constraint)들을 의미한다. 예를 들면, <그림 1>은 6개의 candidate 해 중에서 7가지 조건을 모두 만족시킬 수 있는 해를 뽑는 문제라고 할 수 있다. 이 중에서 F는 2번과 7번 조건만을 만족하는 candidate 해이다.

**- Dancing Link**

Dancing Link란 Algoritm X를 위해 사용되는 자료구조이며, Doubly Linked List로 특별한 매트릭스를 구현한 것이다. Doubly Linked List의 head 노드는 열(column)을 나타내는 리스트를 가리키게 된다. 조건이 7가지인 경우라면, <그림 2>와 같이 head 노드는 7개의 노드를 가진 list의 head가 된다. head 노드를 통해 참조할 수 있는 각 열(column) 노드들은 각 조건을 만족하는 candidate 해를 상징하는 row에 대하여 노드를 만들 수 있고, 한 row 이내의 모든 노드들 또한 Doubly Linked Lisk로 관리된다. 즉, Dancing Link의 각 노드는 4개의 link를 가지며 2개는 column에 대한 link로서, 2개는 row에 대한 link로서 관리된다. <그림 2>의 첫 번째 row는 C, E, F 조건을 만족하는 candidate 해를 나타낸 다는 것을 알 수 있다.

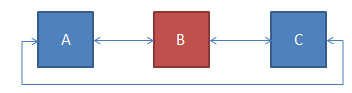


<그림 2>

**- DLX의 장점**

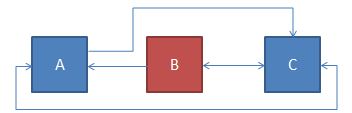
DLX의 장점은 각 노드가 Doubly Linked List로 구현되어 있으므로, 삭제와 복구 연산이 쉽다는 것이다. 이는 DLX의 독특한 자료 구조 덕분에 가능한 것이다. DLX에서는 삭제 연산을 “Cover”라 하고, 복구 연산을 “Uncover”라 부르고 있다. Cover와 Uncover 연산을 통해 Backtracking을 빠른 시간 안에 할 수 있다.

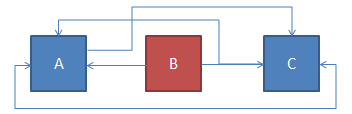
**- Cover 연산**



<그림 3>

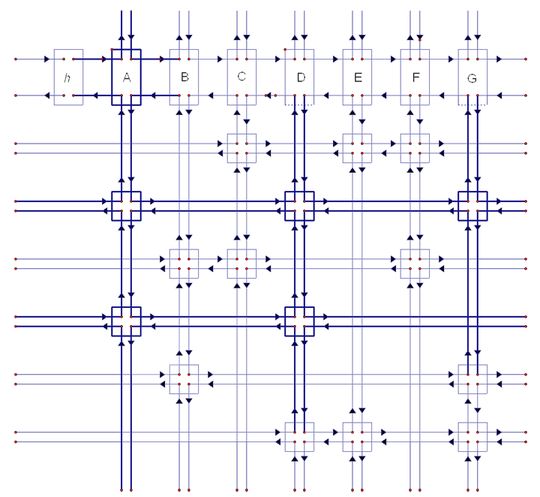
<그림 3> 과 같이 이루어진 Doubly Linked List에서 B 노드를 삭제할 때에는 단 두 개의 명령문 만이 필요하다. 즉 <그림 4>와 같이 B를 가리키고 있던 A의 right link를 C를 가리키게 하고, B를 가리키고 있던 C의 left link를 A를 가리키게 하면 된다. 따라서 B 노드를 삭제하기 위해서 O(n) 시간밖에 걸리지 않는다.





<그림 4>

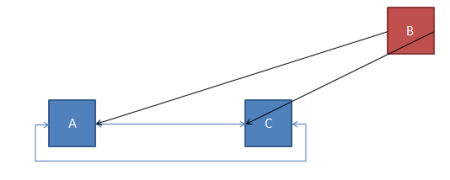
Dancing Link에서 Cover 연산은 열(Column) 노드에 대하여 적용하는 연산이며, 특정 열 노드가 가리키는 모든 노드와 그 노드가 포함된 row의 노드를 모두 삭제하는 연산이다. <그림 5>는 열 노드 A에 대하여 Cover 연산을 수행하는 것을 보여준다. 짙은 색으로 표시된 노드와 link는 Cover로 인해 삭제될 것이다.



<그림 5>

**- Uncover 연산**

삭제된 B 노드를 다시 리스트에 복구할 때에도 단 두 개의 명령문 만이 필요하다. 즉 <그림 6>와 A와 C의 link를 Cover와는 반대로 바꿈으로써 Uncover 연산을 수행한다. 따라서 B 노드를 복구하기 위해서 O(n) 시간밖에 걸리지 않는다.



<그림 6>

Dancing Link에서 Uncover 연산은 열(Column) 노드에 대하여 적용하는 연산이며, 특정 열 노드에 대하여 이루어졌던 Cover 연산을 복구하는 연산이다.

### **Levenshtein distance**

루트 노드에서 리프 노드(leaf node)까지의 경로는 해답후보(candidate solution)가 되는데, 깊이우선검색을 하여 그 해답후보 중에서 해답을 찾을 수 있다. 그러나 이 방법을 사용하면 해답이 될 가능성이 전혀 없는 노드의 후손노드(descendant)들도 모두 검색해야 하므로 비효율적이다.

그래서 노드의 유망성을 확인한다. 전혀 해답이 나올 가능성이 없는 노드는 유망하지 않다(non-promising)고 하고(해당 노드의 서브트리 검색은 무의미), 그렇지 않으면 유망하다(promising)고 한다. 어떤 노드의 유망성을 점검한 후, 유망하지 않다고 판정이 되면 그 노드의 부모노드(parent)로 돌아간다.(“Backtrack”). 그런 다음, 다시 후손노드에 대한 검색을 계속하게 되는 절차이다.

Backtracking 알고리즘은 상태공간트리에서 깊이우선탐색을 실시하는데, 유망하지 않은 노드들은 가지쳐서(pruning) 검색을 하지 않으며, 유망한 노드에 대해서만 그 노드의 자식노드(children)를 검색한다.

이 알고리즘은 다음과 같은 절차로 진행된다.

1. 상태공간트리의 깊이우선탐색을 실시한다.

2. 각 노드가 유망한지를 점검한다.

3. 만일 그 노드가 유망하지 않으면, 그 노드의 부모노드로 돌아가서 검색을 계속한다.

### **Wagner-Fischer algorithm**

루트 노드에서 리프 노드(leaf node)까지의 경로는 해답후보(candidate solution)가 되는데, 깊이우선검색을 하여 그 해답후보 중에서 해답을 찾을 수 있다. 그러나 이 방법을 사용하면 해답이 될 가능성이 전혀 없는 노드의 후손노드(descendant)들도 모두 검색해야 하므로 비효율적이다.

그래서 노드의 유망성을 확인한다. 전혀 해답이 나올 가능성이 없는 노드는 유망하지 않다(non-promising)고 하고(해당 노드의 서브트리 검색은 무의미), 그렇지 않으면 유망하다(promising)고 한다. 어떤 노드의 유망성을 점검한 후, 유망하지 않다고 판정이 되면 그 노드의 부모노드(parent)로 돌아간다.(“Backtrack”). 그런 다음, 다시 후손노드에 대한 검색을 계속하게 되는 절차이다.

Backtracking 알고리즘은 상태공간트리에서 깊이우선탐색을 실시하는데, 유망하지 않은 노드들은 가지쳐서(pruning) 검색을 하지 않으며, 유망한 노드에 대해서만 그 노드의 자식노드(children)를 검색한다.

이 알고리즘은 다음과 같은 절차로 진행된다.

1. 상태공간트리의 깊이우선탐색을 실시한다.

2. 각 노드가 유망한지를 점검한다.

3. 만일 그 노드가 유망하지 않으면, 그 노드의 부모노드로 돌아가서 검색을 계속한다.

## **수행 내용**

### **Backtracking**

2.1.1의 연구내용에서 살펴본 backtracking 알고리즘 수행 절차에 따라 수행된다.

우선 풀어야 하는 Sudoku 문제 개수를 위한 numCases, Sudoku 판의 행과 열을 위한 row와 col, Sudoku 판을 위한 sudokuMap이라는 이름의 2차원 배열을 선언한다.

그리고 문제를 풀기 위한 상태로 초기화를 한다.

findAnswer(sudokuMap, 0, 0,)으로 문제를 풀기 시작한다. Sudoku 판의 왼쪽 최상단 부분의 숫자부터 채우기 때문에 0, 0이 인자로 전달된다. findAnswer()에서 순서는 이미 숫자가 채워졌다면 다음 열을 보고, 다음 행을 본다.

숫자가 채워지지 않았다면 그 칸에 1부터 9까지 모든 숫자 중에서 어떤 숫자가 유망한 숫자 인지를 확인한다. 여기서 유망하다는 것은 promising(sudokuMap, row, col, num)을 수행하여 1이 반환되어 참이 되는 경우를 말한다. Sudoku 규칙에 따라서 같은 행, 열에 중복된 숫자가 이미 있는지 확인하는 것이다. 행과 열을 확인하는데 그치지 않고 3X3 영역에 대해서도 확인을 한다.

숫자를 채워나가는 도중에 어느 숫자 현재 위치에서 유망하지 않은 경우가 생긴다. 이런 경우 실패한 게 아니라면, 다시 이전 단계로 돌아가서(backtracking) 다른 숫자로 채워나간다.

아래의 input값은 3개의 Sudoku 문제이다. 이를 입력 받아서 실행해보았다.

6 0 7 0 0 0 8 0 1 5 3 0 0 7 0 0 0 0 0 0 5 4 2 9 3 0 0

0 0 5 8 0 1 3 0 0 6 0 0 1 9 5 0 0 0 0 7 0 0 0 0 0 6 0

0 0 0 4 0 3 0 0 0 0 9 8 0 0 0 0 6 0 0 9 8 0 0 0 2 4 0

2 0 0 5 0 4 0 0 3 8 0 0 0 6 0 0 0 3 0 0 0 2 3 1 0 0 0

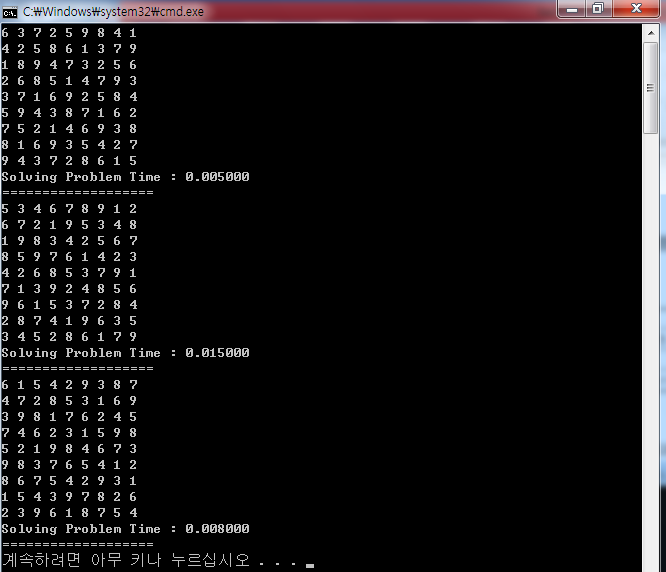
0 7 0 0 0 0 0 8 0 4 0 0 8 0 3 0 0 1 0 0 0 0 0 0 0 0 0

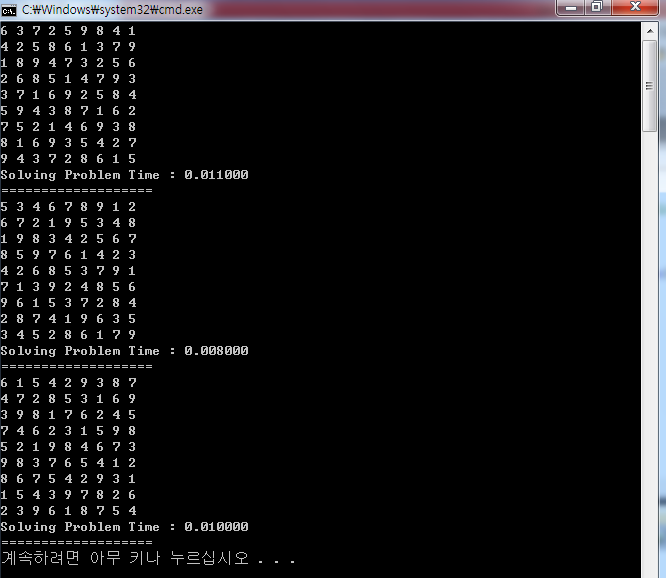
5 0 0 3 0 7 0 0 2 7 0 0 0 2 0 0 0 6 0 0 0 7 6 5 0 0 0

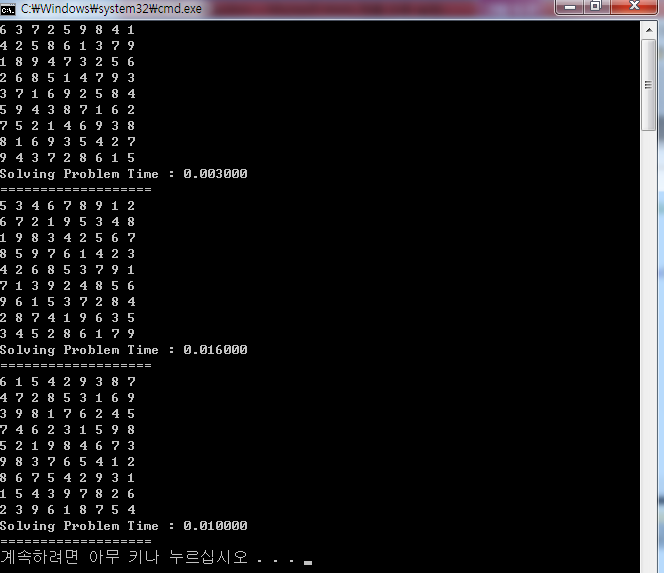
0 0 0 1 0 6 0 0 0 0 6 0 0 0 0 2 8 0 0 6 7 0 0 0 9 3 0

0 0 6 9 0 5 4 0 0 0 0 0 4 1 9 0 0 5 0 5 0 0 0 0 0 2 0

9 0 3 0 0 0 6 0 5 0 0 0 0 8 0 0 7 9 0 0 9 6 1 8 7 0 0







이와 같이 총 3번의 실행 결과마다 문제 풀이 시간은 다르지만 빠른 시간 내에 문제 풀이가 수행 됨을 알 수 있다.

9X9 Sudoku 문제를 해결하는 데 문제가 없고, 결과값이 바르게 나오는 것을 확인할 수 있다. 이 정도 크기에서는 컴파일과 동시에 결과가 출력이 되었다. 효율적으로 Sudoku를 풀 수 있는지 확인하기 위해서 input의 크기를 늘려보았다.

16x16의 Sudoku로 변경 하고 (size를 9에서 16로, 문제 해결 부분과 조건 확인 부분을 16x16일 경우로 수정) 16x16 스도쿠 문제를 해결해 보도록 하였다.

<16x16 스도쿠 문제의 input 값>

12 2 0 7 0 0 0 16 6 15 0 0 0 3 0 14

0 0 0 9 12 0 2 0 0 0 0 0 16 1 0 6

0 11 0 13 0 0 7 0 4 0 8 0 0 0 15 0

0 8 0 3 0 0 1 0 0 0 16 13 11 0 4 2

0 0 15 0 0 0 0 0 1 6 11 0 0 0 3 0

3 0 0 0 0 0 0 0 0 4 0 16 8 0 6 12

0 0 0 14 2 0 0 0 0 12 0 0 7 4 0 0

0 12 8 4 10 0 0 14 7 0 9 3 15 0 0 0

0 0 0 16 3 7 0 4 5 0 0 9 10 14 13 0

0 0 13 8 0 0 11 0 0 0 0 12 4 0 0 0

4 9 0 5 15 0 14 0 0 0 0 0 0 0 0 11

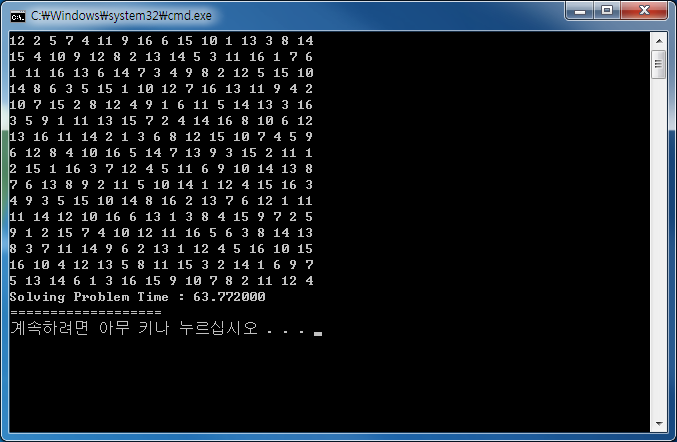
0 14 0 0 0 6 13 1 0 0 0 0 0 7 0 0

9 1 0 15 7 4 0 0 0 16 0 0 3 0 14 0

0 3 0 0 0 9 0 2 0 1 0 0 5 0 10 0

16 0 4 12 0 0 0 0 0 3 0 14 1 0 0 0

5 0 14 0 0 0 16 15 9 0 0 0 2 0 12 4



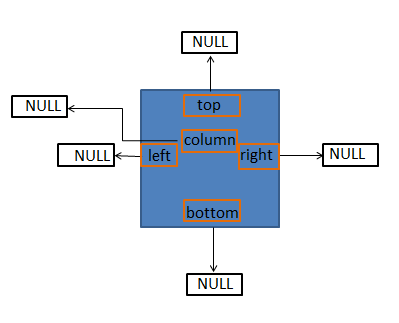
위의 그림처럼 결과를 도출해 내는 데에 9X9의 크기 일 때보다 많은 시간이 소요되어 결과값이 바로 출력되지 않았다.

따라서 수행시간을 줄이고 연산 과정을 최소화하여 프로그램의 효율성을 높여야 하겠다고 생각, Sudoku 문제를 Backtracking이 아닌 다른 방법을 활용하여 구현해야 함을 확인하였다.

### **Dancing Link**

Sudoku 문제 해결을 위해서는 Sudoku에 맞는 매트릭스가 필요하다. Dancing Link에서 각 행(row)은 candidate 해를 나타낸다고 했다. 9행 9열로 이루어진 Sudoku 문제의 candidate 해는 “n행 m열에 값 r이 들어간다”라고 할 수 있다. 이때, n, m, r 모두 1, 2, 3, …, 9의 범위를 가지므로 Sudoku를 위한 Dancing Link의 row는 총 729() 개이다. Dancing Link에서 각 열(column)은 조건(constraint)을 나타낸다고 했다. Sudoku 문제의 조건은 하나의 행, 하나의 열, 하나의 박스에는 1부터 9까지 숫자로 이루어져야 하며, 모든 cell에는 값이 있어야 한다는 것이다. 행, 열, 박스는 총 9개이고, 9x9 Sudoku 문제에서 cell은 총 81개 이므로, Sudoku를 위한 Dancing Link의 column은 총 324(81 \* 4) 개이다.

각 노드는 <그림 7>과 같이 총 5개의 link를 갖는다. 2개의 link는 row를 나타내는 doubly linked list를 구현하기 위한 left, right이다. 2개의 link는 column을 나타내는 doubly linked list를 구현하기 위한 top, bottom이다. 1개의 link는 현재 노드가 어느 조건(constraint)에 해당하는 노드인지를 표시하기 위한 column이라는 link이다.



<그림 7>

Dancing Link를 사용하여 수도쿠 문제를 해결하는 내용을 Pseudo Code로 나타내면 다음과 같은 흐름을 보인다.

Init Dancing Link for Sudoku

numGivenSolution = Parse\_grid

Search(DL\_board, numGivenSolution)

**- Init**

우선 Sudoku 조건에 맞는 Dancing Link를 구현해야 한다. Sudoku 문제를 풀기 위한 Dancing Link는 729개의 row와 324개의 column으로 구성되어 있다. 또한 각 row에 대하여 조건에 맞게 노드를 생성하는 작업까지 이루어져야 한다.

**- Parse\_grid**

Input 파일로부터 Sudoku 보드를 읽어 들인다. 이미 특정 cell에 숫자가 적혀있는 경우, solution으로서 기록해야 할 필요가 있다. 이에 대한 작업을 수행하며, 주어진 solution에 대하여 Cover 연산을 수행한다. Cover 연산을 통해 주어진 solution과 겹치는 조건의 다른 candidate 해를 후보에서 제거할 수 있다. 이 과정을 통해 backtracking 할 때, 탐색할 후보가 줄어드는 효과를 기대할 수 있다. 모두 수행한 후 주어진 solution의 개수를 반환한다.

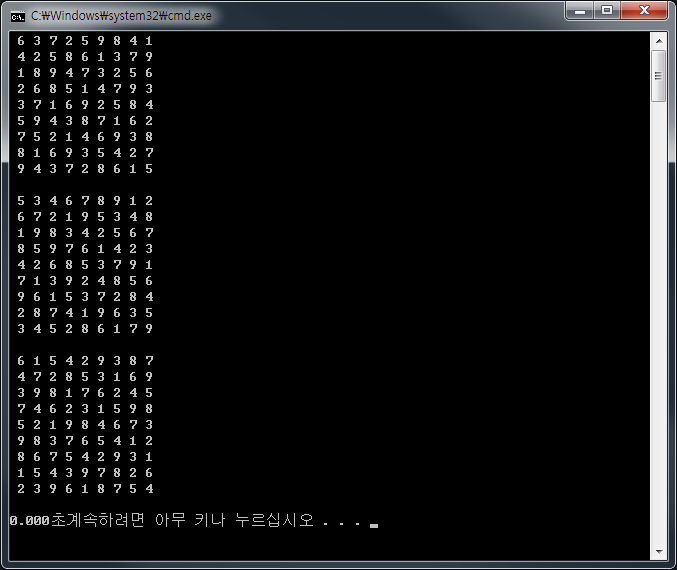
**- Search**

실질적으로 backtracking이 일어나는 부분이다. 재귀적인 방법으로 수행된다. 아직 Cover 되지 않은 column을 선택하고, 선택된 column에 대하여 cover를 수행한다. 모든 column이 Cover 되어 선택될 column이 없다면 Sudoku 문제를 해결한 것이므로, Search를 종료하고 solution을 출력한다. Cover 연산 후에 남은 row 중 하나를 solution으로서 선택한 후, solution으로 선택된 row에 포함된 노드에 대하여 Cover를 수행한다. 재귀적으로 Search를 호출하여 반복한다. 이때, solution으로 선택된 row가 Sudoku 문제를 해결할 수 없다고 판단되면, Uncover 연산을 수행하여 Backtracking을 한다.

실행 시간이 0.000초로 출력되는 것을 알 수 있다.

여러 번 돌리는 경우 0.015초나 0.016초가 나오지만 이건 환경적인 문제라고 생각이 된다.

0.000초가 지속적으로 나오는 것으로 보아 앞에서 구현했던



# **수정된 연구내용 및 추진 방향**

## **수정사항**

앞에서 9X9 크기의 문제에 대해서 Dancing Link Algorithm을 적용하여 Sudoku 문제를 해결하는 프로그램의 속도가 더 빠름을 확인할 수 있었다. Backtracking으로 구현한 프로그램은 입력 값의 크기가 커질수록 느려지는 것을 확인했다. 하지만 Dancing Link Algorithm을 적용하여 문제를 해결하는 경우에 아직 16X16을 대상으로 수행하여 그 시간을 확인 하지 않았다. Backtracking으로 구현한 프로그램은 입력 값의 크기가 커질수록 느려지는 속도의 차이가 Dancing Link Algorithm과 비교해서 얼마나 되는지를 확인하기 위해서 소스 코드에 약간의 수정을 하여 시간을 비교해보는 것으로 더 확실히 할 것이다.

# **부록**

## **Backtracking**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define ROW\_SIZE 16

#define COL\_SIZE 16

//promising인지 확인한다.

int promising(int sudokuMap[][COL\_SIZE], int row, int col, int num){

int i,j;

int rowStart=0;

int colStart=0;

//현재 넣고자하는 숫자 num 이 가로, 세로에서 한번 이상 나왔는지 확인한다.

for(i=0; i<16; i++)

{

if((sudokuMap[row][i] == num) || (sudokuMap[i][col] == num)) // 한 번이라도 그 숫자가 나온다면, 그 num은 non promising 이므로, 0을 반환한다.

{

return 0;

}

}

//1 Region 안에서 같은 숫자가 두 번 나오는지 확인한다. num 이 나오기만 하면, 중복이므로, sudoku에 숫자를 채울 수 없다.

rowStart = (row/4)\*4;

colStart = (col/4)\*4;

for(i=rowStart; i<rowStart+4; i++)

{

for(j=colStart; j<colStart+4; j++)

{

if(sudokuMap[i][j] == num) // 한 번이라도 그 숫자가 나온다면, 그 num은 non promising 이므로, 0을 반환한다.

return 0;

}

}

// 위의 조건을 다 통과한다면, promising 하므로, 1을 반환한다.

return 1;

}

//스도쿠판을 채워간다.

int findAnswer(int sudokuMap[][COL\_SIZE], int row, int col){

int i;

//row와 col의 범위안에 있다면,

if(row < ROW\_SIZE && col < COL\_SIZE){

//sudoku 칸이 0이 아니면, 이미 채워져있는 것이므로, 다음 빈칸을 찾을 때까지 sudoku를 진행한다.

if(sudokuMap[row][col]!= 0){

//다음 칸이 열을 벗어나지 않았다면, findAnswer를 col+1방향으로, recursive하게 호출한다.

if(col+1 < COL\_SIZE)

return findAnswer(sudokuMap, row, col+1);

//다음 칸이 행을 벗어나지 않았다면, findAnswer를 row+1방향으로, recursive하게 호출한다.

else if(row+1 < ROW\_SIZE)

return findAnswer(sudokuMap, row+1, 0);

//다음 칸이 행과 열을 벗어났다면, 다 끝난 것이므로 return 한다.

else

return 1;

}

else{ //sudoku 칸이 0이면, 그곳에 들어갈 숫자를 찾아야 한다.

for(i=0;i<16;i++){ //1부터 9까지 돌면서 원하는 숫자를 찾는다.

if(promising(sudokuMap, row, col, i+1)){ // 현재 i+1(원하는 숫자) 가 promising 하면, 숫자를 채운다.

sudokuMap[row][col] = i+1;

// 숫자를 채운 후에도 아직 범위를 벗어나지 않았다면, recursive하게 호출한다.

if(col+1 < COL\_SIZE){

if(findAnswer(sudokuMap, row, col+1)) // 현재 행에서 col만 1씩 증가시키며, sudoku를 채운다.

return 1;

else // sudoku를 채울 수 없었다면, 그 자리를 다시 빈칸상태로 복구한다.

sudokuMap[row][col] = 0;

}

else if(row+1 < ROW\_SIZE){

if(findAnswer(sudokuMap, row+1, 0)) // 다음 행에서 출발해야 하므로, row+1 해주고, 새로운 열부터 다시 시작해야 하므로, col 에는 0을 넣어준다.

return 1;

else // sudoku를 채울 수 없었다면, 그 자리를 다시 빈칸상태로 복구한다.

sudokuMap[row][col] = 0;

}

else // 숫자를 채우고 났는데 범위를 벗어났다는 것은 sudoku를 다 채웠다는 뜻이므로, True를 반환한다.

return 1;

}

}

}

//sudoku 칸이 0이면서, 그곳에 들어갈 숫자도 찾지 못한 것이므로, False를 반환한다.

return 0;

}

else{ //범위를 벗어났다면 sudoku를 다 채운 것이므로, 1을 반환한다.

return 1;

}

}

//main 시작

void main(void){

int i=0,j=0;

//sudoku 문제 갯수

int numCases ;

//행, 열 변수

int row, col ;

//sudoku 판 변수

int sudokuMap[ROW\_SIZE][COL\_SIZE] ;

clock\_t start, end;

//input.txt에서 문제를 입력받는다.

FILE \*fp = fopen("input.txt","r");

if(fp ==NULL)

exit(1);

//sudoku 테스트 케이스의 갯수 입력받는다.

fscanf(fp, "%d",&numCases);

//테스트 케이스만큼 반복

for(i=0;i<numCases;i++){

//스도쿠판 초기화

for(row=0;row<ROW\_SIZE;row++)

{

for(col=0;col<COL\_SIZE;col++)

{

fscanf(fp,"%d",&sudokuMap[row][col]);

}

}

//스도쿠 문제를 푼다. 0,0부터 sudoku를 채운다. 0,0부터 다 채워졌다면, sudoku를 출력한다.

start = clock ();

if(findAnswer(sudokuMap,0,0))

{

for(row=0;row<ROW\_SIZE;row++)

{

for(col=0;col<COL\_SIZE;col++)

printf("%d ", sudokuMap[row][col]);

printf("\n");

}

}

else //그렇지 않으면, sudoku를 풀지 못한 경우이다.

printf("\n\nWe can't find solution!!\n\n");

end = clock ();

printf ("Solving Problem Time : %f \n", ((double)(end - start)) / CLOCKS\_PER\_SEC);

printf("===================\n");

}

}

## **Dancing Link**

#include <Windows.h>

#include <limits.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define D 3 // dimension of sudoku

#define N (D\*D) // number of row, column, box

/\*\*\*

Struct: node

represents node in dancing link data structure.

has 5 links: left, right(for row); up, down(for column), column(for constraints)

has union value for soulution

\*\*\*/

struct node {

union {

char size; // headers exclusively use size (to find most interesting column)

uint32\_t name; // cells exclusively use name (to print solution)

};

struct node \*left, \*up, \*right, \*down;

struct node \*column;

};

/\*\*\*

enumerator for determining position of node

; cell, row, column, box

\*\*\*/

enum { CST\_CELL, CST\_ROW, CST\_COL, CST\_BOX, CST\_N };

#define NODES\_HEIGHT (N+1) /\* +1 for header \*/

#define NODES\_WIDTH (CST\_N \* N\*N)

/\*\*\*

Struct: sudansu

struct for sudoku solution.

has head node and node list for recording solution

has row list for search(cf. Search function)

\*\*\*/

struct sudansu {

uint32\_t solution[N\*N];

struct node h;

struct node nodes[NODES\_HEIGHT \* NODES\_WIDTH];

struct node \*rows[N\*N\*N];

};

/\*\*\*

Function: get\_node\_position

determine position of node and return specific unsigned number

\*\*\*/

static unsigned get\_node\_position(unsigned type, unsigned r, unsigned c, unsigned z)

{

switch (type) {

case CST\_CELL: return c + r\*N;

case CST\_ROW: return z + r\*N;

case CST\_COL: return z + c\*N;

case CST\_BOX: return z + c / D\*N + r / D\*D\*N;

default: abort();

}

}

#define CELL\_NAME(row, col, numchr) ((row)<<16 | (col)<<8 | (numchr))

/\*\*\*

Function: init

init dancing link board for sudoku problem.

Param: ctx pointer of sudansu structure. for being initialized

\*\*\*/

static void init(struct sudansu \*ctx)

{

unsigned i, j, r, c, z, cst\_type;

struct node \*x, \*row\_nodes[CST\_N]; // one node per line per constraint type

memset(ctx, 0, sizeof(\*ctx));

/\* vertical links \*/

x = ctx->nodes;

for (j = 0; j < NODES\_HEIGHT; j++) {

for (i = 0; i < NODES\_WIDTH; i++) {

x->up = j == 0 ? &ctx->nodes[(NODES\_HEIGHT - 1)\*NODES\_WIDTH + i] : x - NODES\_WIDTH;

x->down = j == NODES\_HEIGHT - 1 ? &ctx->nodes[0 \* NODES\_WIDTH + i] : x + NODES\_WIDTH;

x->column = &ctx->nodes[i];

x++;

}

}

/\* header horizontal links \*/

ctx->h.right = ctx->nodes;

ctx->h.left = ctx->nodes + NODES\_WIDTH - 1;

for (i = 0; i < NODES\_WIDTH; i++) {

ctx->nodes[i].left = i == 0 ? &ctx->h : &ctx->nodes[i - 1];

ctx->nodes[i].right = i == NODES\_WIDTH - 1 ? &ctx->h : &ctx->nodes[i + 1];

}

/\* nodes horizontal links \*/

i = 0;

for (r = 0; r < N; r++) {

for (c = 0; c < N; c++) {

for (z = 0; z < N; z++) {

/\* locate the CST\_N nodes per row \*/

for (cst\_type = 0; cst\_type < CST\_N; cst\_type++) {

unsigned col = get\_node\_position(cst\_type, r, c, z);

struct node \*colhead = &ctx->nodes[cst\_type\*N\*N + col];

// locate first unused node of the colum

x = colhead->down + NODES\_WIDTH \* colhead->size++;

x->name = CELL\_NAME(r, c, z + '1');

row\_nodes[cst\_type] = x;

}

/\* link them together \*/

for (cst\_type = 0; cst\_type < CST\_N; cst\_type++) {

row\_nodes[cst\_type]->left = row\_nodes[cst\_type == 0 ? CST\_N - 1 : cst\_type - 1];

row\_nodes[cst\_type]->right = row\_nodes[cst\_type == CST\_N - 1 ? 0 : cst\_type + 1];

}

/\* index the row since the row positions in ctx->nodes are not linear \*/

ctx->rows[i++] = row\_nodes[0];

}

}

}

}

/\*\*\*

Function: cover

cover the column.

Param: c pointer of node structure for being covered

\*\*\*/

static void cover(struct node \*c)

{

struct node \*i, \*j;

c->right->left = c->left;

c->left->right = c->right;

for (i = c->down; i != c; i = i->down) {

for (j = i->right; j != i; j = j->right) {

j->down->up = j->up;

j->up->down = j->down;

j->column->size--;

}

}

}

/\*\*\*

Function: uncover

uncover the column.

Param: c pointer of node structure for being uncovered

\*\*\*/

static void uncover(struct node \*c)

{

struct node \*i, \*j;

for (i = c->up; i != c; i = i->up) {

for (j = i->left; j != i; j = j->left) {

j->column->size++;

j->down->up = j->up->down = j;

}

}

c->right->left = c->left->right = c;

}

/\*\*\*

Function: print\_solution

print solution and return 1 for informing end of search

\*\*\*/

static int print\_solution(const uint32\_t \*solution)

{

unsigned i, j, k;

char grid[N\*N] = { 0 };

const char \*p = grid;

for (i = 0; i < N\*N; i++)

{

const char row = solution[i] >> 16 & 0xff;

const char col = solution[i] >> 8 & 0xff;

const char num = solution[i] & 0xff;

grid[row\*N + col] = num;

}

for (j = 0; j < N; j++) { /\* print the solution board(apply the given form) \*/

for (i = 0; i < N; i++) {

printf(" %c", \*p ? \*p : '?');

p++;

}

if (j + 1 != N && (j + 1) % N == 0) {

printf("\n ");

}

printf("\n");

}

printf("\n");

return 1;

}

/\*\*\*

Function: choose\_a\_column

choose a column for search.

if c equals h, the sudoku problem is solved because all constraints are satisfied

Param: c pointer of node structure of head which links constraints node

\*\*\*/

static struct node \*choose\_a\_column(struct node \*h)

{

int size = INT\_MAX;//INT\_MAX : 2147483647

struct node \*j, \*c = h->right;

if (c == h)

return NULL;

for (j = h->right; j != h; j = j->right) /\* choose the most interesting column \*/

if (j->size < size) /\* traveling to find the node which has smallest node->size\*/

c = j, size = j->size;

return c;

}

/\*\*\*

Function: search

choose a column in head node for satisfying constraint.

if chosen column is NULL, all constraints are satisfied: print solution

if chosen column is not NULL, cover the column

and choose a row expected to be included in solution,

cover about this row.

if this row is determined to be not solution,

uncover about row and choose another row(backtracking)

Param: ctx pointer of sudansu structure that has information of sudoku board

k number of solutions in sudoku board that is given by input file

return: if search is successful, return 1

if not successful, return 0

\*\*\*/

static int search(struct sudansu \*ctx, unsigned k)

{

struct node \*c, \*r, \*j;

c = choose\_a\_column(&ctx->h); /\* step 1: chose a column \*/

if (!c) return print\_solution(ctx->solution);

cover(c); /\* step 2: cover by the column chosen at step1 \*/

for (r = c->down; r != c; r = r->down) /\* step 3: travel the sudoku board via Dancing Link \*/

{

ctx->solution[k] = r->name; /\* 3-1: mark the node name to solution board \*/

for (j = r->right; j != r; j = j->right) /\* 3-2: covering \*/

cover(j->column);

if (search(ctx, k + 1)) /\* 3-3: recursive call(then find next value which holds sudoku rule \*/

return 1; /\* if find perfect solution, then finish whole progress \*/

ctx->solution[k] = 0; /\* step 4: if this step is excuting, It is necessary to backtracking\*/

for (j = r->left; j != r; j = j->left) /\* so doing uncover \*/

uncover(j->column);

}

uncover(c);

return 0;

}

/\*\*\*

Function: parse\_grid

input from file, and cover about given solution

Param: ctx pointer of sudansu structure that has information of sudoku board

stream file stream for input

return: if input error, return -1

if parse is successful, return number of given solution

\*\*\*/

static int parse\_grid(struct sudansu \*ctx, FILE \*stream)

{

char buf[16];

unsigned i, j, l,k = 0;

for (j = 0; j < N; j++)

{

for(l = 0; l<N; l++) /\* read initial conditions for each row via the stream \*/

{

fscanf(stream,"%s",&buf[l]);

if(buf[l] == -1)

return -1;

}

for (i = 0; i < N; i++) /\* setting the solution board which is covered by all initial conditions \*/

{

if (buf[i] >= '1' && buf[i] <= '1' + N)

{

struct node \*x, \*r = ctx->rows[j\*N\*N + i\*N + buf[i] - '1'];

ctx->solution[k++] = CELL\_NAME(j, i, buf[i]);

cover(r->column);

for (x = r->right; x != r; x = x->right)

cover(x->column);

}

}

}

return k;

}

int main(int ac, char \*\*av)

{

int k = 0, ncase = 0, loop = 0;

struct sudansu ctx;

FILE \*in\_file = NULL;

DWORD Time1, Time2;

Time1 = GetTickCount();

in\_file = fopen("input.txt","r");

fscanf(in\_file,"%d",&ncase); /\*read the number of cases in "input.txt"\*/

for(loop = 0; loop<ncase; loop++)

{

init(&ctx); /\* init sudoku board \*/

k = parse\_grid(&ctx, in\_file); /\* enter the initial condition of sudoku quiz\*/

search(&ctx, k); /\* serch for solution(recursive)\*/

}

Time2 = GetTickCount();

printf("%.3f초", (Time2-Time1)/1000.0);

return 0;

}

**- 참고**

<그림 1> <http://en.wikipedia.org/wiki/Exact_cover>

<그림 2> <http://arxiv.org/pdf/cs.DS/0011047.pdf>

<그림 5> <https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html>

소스 http://blog.pkh.me/p/9-dlx-and-reduced-memory-footprint.html