Problem Space

States

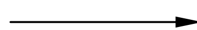
Presentation

我們以一串由 0 到 N 的整數所排成的序列作爲 N-Puzzle 的內部表徵，每個整數只能出現一次，且以 0 表達 blank block。
以 8-Puzzle 爲例：

initial state

Case1:

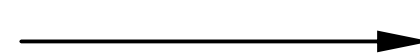
2	4	3
1		6
7	5	8



State = { 2,4,3,1,0,6,7,5,8 }

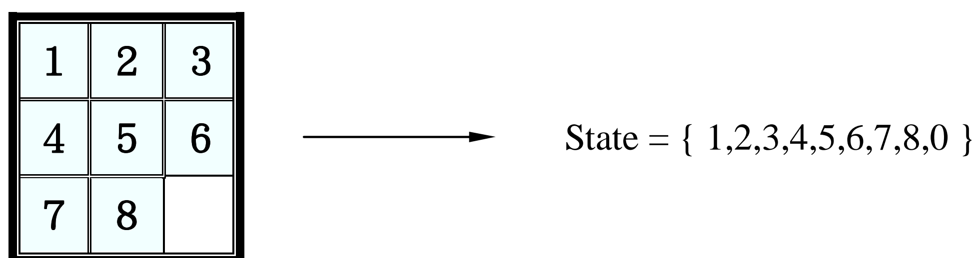
Case 2:

5	4	2
8		3
1	7	6



State = { 5,4,2,8,0,3,1,7,6 }

goal state



Auxiliary Attribute

爲了方便,除了上述的序列外,每個 state 還額外紀錄了長和寬的格子數及 blank block 的 (x,y) 座標。
以 8-Puzzle，其座標由左上的 (0,0) 延伸到右下的 (2,2)

Operators

爲了簡化問題，我們將各個 block 在 board 利用單一的 container 移動位置；以相反的方式看成是這個 blank 在 board 上游走。
所以我們可以很簡單的將 operator 簡化成以下四種。

move blank Up

blank 和緊鄰其上的 block 交換位置。

move blank Down

blank 和緊鄰其下的 block 交換位置。

move blank Left

blank 和緊鄰其左的 block 交換位置。

move blank Right

blank 和緊鄰其右的 block 交換位置。

Solution

Solution 以一串 operators 的 list 表達
operator set = { u,d,l,r } ，分別代表 move blank up, down, left 及 right。
以 case1 爲例，其中一組可行的 solution = { u, l, d, r, d, r }。

Algorithms

General Search Algorithm

提供一個利用搜尋手段解題時的一般性架構，其類 C++的虛擬碼如下：

```
search( Node& node)
{
  pushInitialState();

  for (;;) {
    if (isPoolEmpty())
      return false;

    node= pop();

    if (goalTest(node))
      return true;

    expandAndMerge( node );
  }
}
```

reference

File: GSA.h

Breadth-First Search

pop order

FIFO

expand

expand 時， operators 順序如下：

1. move Blank Up
2. move Blank Down
3. move Blank Left
4. move Blank Right

merge

利用一個 FIFO 的 queue 將 nodes 丟進去。

reference

File: BFSPuzzle.h
File: BFSPuzzle.cc

Depth-First Search

pop order

FILO

expand

expand 時，operators 順序如下：

1. move Blank Up
2. move Blank Down
3. move Blank Left
4. move Blank Right

merge

利用一個 FILO 的 stack 將 nodes 丟進去。

reference

File: DFSPuzzle.h
File: DFSPuzzle.cc

Repeated States Avoiding

strategy

簡單地將所有經歷的 state 丟進 set 裡，因而可避免 global cycles。

reference

File: Puzzle.h
File: Puzzle.cc

Software Building

Analysis & Design

the goal

Correctness 最起碼要能通老師指定的兩個 case 的試驗。
Clarity 系統架構設計要清晰、簡單、易理解，以方便對軟體作追蹤、審閱、除錯、細部調整、功能刪減等。
Performance 在滿足了 correctness 及 clarity 之後，還要考量程式的高階效率和低階效率。舉凡底層資料結構的選擇、和高層抽象表示法連結上的介面設計等，都是考量重點。
Flexibility 我們在設計時還希望對系統架構能保持一定的彈性。如，不要只針對 8-Puzzle 設計，而更進一步粹取出 N-Puzzle 的表達方式。

packages & classes

本系統架構在最高階上簡單區分成 AIUtility、Utility 及 NPuzzle 等三個 package。所有 AI 領域上，通用的概念元件或可共用的演算法都擺在 AIUtility 裡；和 N-Puzzle 相關的東西都擺在 NPuzzle 這個 pacakge 中；而可在多系統間共用的的工具則擺在 Utility 裡：

Package Name	Class Name	Description
Utility	SliceIter	用來輔助，高效率地處理 state 的內部結構
	CSliceIter	用來輔助，高效率地處理 state 的內部結構
AIUtility	GSA	對應到 General search algorithm
NPuzzle	Puzzle	用來支援通用的 NPuzzle 特性
	State	用來維護、管理 Puzzle 的 states 及其 operators
	Node	State 在丟到 Pool 前，要先以 Node 封裝起來
	BFSPuzzle	實作 NPuzzle 問題的 breadth-first search 解法
	DFSPuzzle	實作 NPuzzle 問題的 depth-first search 解法

reference

File: NPuzzleUML.pdf

Implement

language & platform

由於分析及設計階段就是採用 OOA/OOD 的方式，所以很自然地要採用支援 OO 的程式語言來實作。經討論，我們決定採用 ANSI C++來實作這份設計，並利用 gnu 的編譯環境來製作可執行檔。

雖然我們只在

Windows 下的 DJGPP 搭配 rhide 的 IDE 及

Linux 下的 g++搭配 make

試著編譯過我們的源碼。

但理論上應可以適用於任何支援“完整”ANSI C++語法的平台。

請注意，我們除了有用到 STL 外，還使用了 namespace，所以古老的 C++編譯環境可能無法順利編出可執行檔。

design mapping

所有在設計階段的 package 都以 C++的 namespace 對應，而所有封存在其中的概念性 class，都以 C++的 class 對應。這幾乎是一對一的關係，所以分析、設計階段的高階結果，可以在實作階段直接引用。

error preventing

對實用的軟體而言，Robustness 是個重要的課題。所有嚴謹的軟體設計都會制定一致性的錯誤處理策略，以便程式在 run time 發生問題時的挽救，及防止 client 的誤用。
這雖然是個不錯的練習，但由於這裡只是個課程 project，在時間不是很充裕的情況下，我們只簡單地以 C 語言就有支援的 assert 敘述，來支援部份 design by contract 的概念。以防止底層的源碼被 client 誤用。

Make Target

在 Linux 的 g++環境下，只要簡單地在提示符號後面鍵入：
  make
就會自動產生 bfs 及 dfs 這兩個執行檔，根據內定，這兩個執行檔是以 case1 爲測試案例，若要試驗別的案例，請自行修改 `BFSTest.cc` 及 `DFSTest.cc` 這兩個源碼中的 `initial state` 及 `goal state`。

Reference

`File: Makefile`

Execution Tracing

Expands Limited

由於老師要求若 expand 的次數超過 100 時還沒跑出解答的話，程式要自動終止，以方便助教 track，所以我們就在 class Puzzle 中以 enum 的方式設了一個 ExpandsLtd 的常數來處理這件事。

Reference

File: Puzzle.h
File: BFSPuzzle.cc
File: DFSPuzzle.cc

BFS

Case1

expands = 73
depth = 6
solution = { u, l, d, r, d, r }

reference

File: bfs1.out (ExpandsLtd==100)

Case2

在 expand 的次數到達 100 時，還沒找到解答，不過我們還是決定放寬其 expand 的限制：

expands = 1974
depth = 12
solution = { l, d, r, u, u, l, d, r, u, r, d, d }

reference

File: bfs2.out (ExpandsLtd==100)
File: bfs2-1.out (ExpandsLtd==10000)

DFS

Case1

在 expand 的次數到達 100 時，還沒找到解答，且放寬限制後還是跑很久，只好宣告放棄：

expands = 100
depth = 99
solution = not found

reference

File: dfs1.out (ExpandsLtd==100)

Case2

在 expand 的次數到達 100 時，還沒找到解答，且放寬限制後還是跑很久，只好宣告放棄：

expands = 100
depth = 99
solution = not found

reference

File: dfs2.out (ExpandsLtd==100)

Conclusion

根據 8-Puzzle 及老師給的兩個 test case 看來，N-Puzzle 問題若光就 BFS 及 DFS 兩種方法比較，比較適合用 BFS 的方式來解。
因爲其可行 solution 的深度都在合理的範圍，但 solution path 和所有可能的 path 比較起來，相對來得很小。