情報実験第一（O）大課題レポート

提出日：July 19, 2012

学科：情報工学科

学籍番号：11_11730 / ログイン名：j11173

氏名： 澤田 賢祐

1　課題１

Program1(ファイル名：report1.c)

1. パズルの読み込み

   基本的には配布された pdf に従った方法を用いているので、特筆すべき箇所はない。コマンドライン引数で渡された入力ファイル名のファイルをオープンし、グローバルに確保した char 型の 36*36 配列にマップ情報を取得する。その際、後述の正誤判定に用いるために X*Y-(盤面の全ての数字の和) の値を求めておく。

   関連する関数

   void load_puzzle(char* filename)

2. 入力を受け取る

   入力された座量に対する処理は以下の４種類である。

   (1)　マップの範囲外である
   (2)　壁のあるマスである　⇒　空白にする
   (3)　空白のマスである　⇒　2x2 ルールを通過すれば壁にする
   (4)　数字のあるマスである

   マップの範囲外かどうかの判定は単なる座標の比較だが、この判定処理は他の場所でもよく使う上バグも発生しやすいので、関数 is_out_of_range(int x, int y) として分離してある。2x2 ルールの適応についても、方向を配列に格納するなどの工夫はしてあるが所詮は、そのマスを L 字に囲むマスの全てが壁であったらそのマスには壁にできない、という処理を四方向に行っているだけである。

   関連する関数

   int input(int* walls)
   int check2x2(int x, int y)

3. 正誤判定

   正誤を判定する条件は以下の２つである

   (1) 壁が全て連結である。

   (2) 全ての島の面積が正しい。

   これらは共に、関数の再帰呼び出しを用いた深さ優先探索 (depth first search) により実現可能である。なお、関数の戻り値は、その引数のマスと結合している、チェック済みでない壁の総数である。探索は隣接する 4 方向について再帰呼び出しによって行われ、(この際に方向を格納した配列を用いると便利である)、もしその位置がマップの範囲外、あるいは壁でないマスであるならば、呼び出し先の関数において棄却され、戻り値は 0 となる。

   正答における壁の数は X*Y-(盤面の全ての数字の和) で不変であるから、main() 内にて現在の壁の数を変数に保持することによって、盤面の正誤の判定において、壁の数が不正である場合はその時点で不正解という判断が下せる。これにより、わざわざ入力毎に探索をして正誤の判定をする必要はなくなり、盤面が正答に近い場合 (=壁の数が正しい場合) においてのみ、比較的重い (と言ってもこの場合大した計算量ではないが) 処理である深さ優先探索による正誤判定を実行することになる。

   関連する関数

int isfinish(int walls)
int search(int i, int j, int type)

2　課題２

Program2(ファイル名：report2.c)

1. ぬりかべをプログラムが解く手法

   ぬりかべをプログラムが解くメソッドとして、以下の自動的処理と仮定的処理が挙げられる。

   ・自動的処理

   仮定を含まず、確信的に決定することができる処理の概要は以下の通りである。当然のことながら、

   「誤りを含まない盤面に対し以下の 10 種類の処理を任意の回数適用した後の盤面は、誤りを含まない。」… (a)

   処理１．　四角潰し

   ルールより、壁が 2x2 以上で並ぶことはない。このことから、L 字型に壁が３つ並んでいたら、L 字の内側の１マスはかならずどこかの島の領土となる。

   処理２．　壁伸ばし

   ルールより、全ての壁は連結でなければならない。よって、孤立している壁郡 (*1) があり、かつその壁が伸ばせるマスが一つしかない場合、そのマスは壁でなくてはならない。
   1. 孤立している壁：　ある連結壁郡に含まれる壁の数が、現在の全体の壁の数を下回るとき、その壁郡を孤立しているという。

   処理３．　数字伸ばし

   ある島の領土数が必要な領土数を満たしておらず、かつ伸ばせるマスが一つしかない場合、そのマスはその島の領地となる。

   処理４．　斜め殺し

   互いに id(*2) の異なる領土が、斜め隣に並んでいた場合、両者が結合することはないため、もう一方の対角の２マスはいずれも壁となる。
   2id：　全ての領土は一つの領主 (数字) に属する。全ての領主は互いに異なる id(通し番号) を持つことから、全ての領土 (∈島) は id を持つ。

   処理５．　間殺し

   互いに id の異なる領土が、一マスを隔てて横あるいは縦に並んでいた場合、両者が結合することはないため、その間のマスは壁となる。

   処理６．　確定領土閉鎖

   領土数が必要な領土数に達した場合、その領土の周囲は壁で囲まれる。

   処理７．　領土伸ばし

処理１．などにより、id を持たない (どの領主に所属するか未定の) 領土 (以降、ノラ領地と表現する) が出現する。そういった領土に関しても、処理３と同様に、伸ばせるマスが一つしかない場合はそのマスも領土となる。また、id を持った領土と隣接していた場合、そのノラ領土の id も判明する。

処理８．　領主探し

ノラ領地が id=x を持った領地に隣接していた場合、そのノラ領地は id=x の島の所属となる。あまり利点を感じないかもしれないが、処理１０などで非常に有効に作用する。

処理９．　到達不可能マス潰し

どの島からも到達することが不可能な位置 (距離) にあるマスは壁となる。この際、あるノラ領地が唯一の島からのみ到達可能ならば、そのノラ領地はその島の所属と考えられる。

処理１０．　全列挙からの確定潰し

ある一つの島が取れる形は有限個のパターンで列挙できる。そのいずれにおいても壁、あるいは領土となるマスにおいては、そのマスの状態が確定する。

・仮定的処理
上記の他に、処理の方法として、あるマスを仮定して矛盾を導くという手法がある。すなわち、

仮定を一回行う度に状態は遷移し、状態番号は 1 だけ増加するとする。状態番号 0 とは仮定が行われていない状態であるとする。状態 n において、あるマス A を壁であると仮定し得られた状態 n+1 に、上記の処理を任意の回数適応させた後、パズルの盤面が誤りを含んでいた場合、(a) より状態 n+1 においてパズルは誤りを含んでいたことがわかる。よって、状態 n+1 から正答にたどり着くことはないから、状態 n において、「マス A は壁である⇒誤答」であると分かる。

この処理を帰納的に繰り返すことにより、最終的には正答にたどり着くことができる。

2. アルゴリズム

それぞれの処理について、具体的にどのようにしてプログラムを組むかについて考えてみる。

処理１．　四角潰し　計算量 O(XY)

これについてはごく naive に、マップ全体を２重ループで見ていけばよいだろう。

処理２．　壁伸ばし　計算量 O(XY)

壁郡から伸ばせるマスが一つしかない、すなわち壁郡の隣接する空白マスが１つしかないかを調べるには、関数の再帰呼び出しによる深さ優先探索によって容易に実現できる。また、壁郡に含まれる壁の数についても、ほとんど同じ手法で求めることができる。また、探索の順序としては、マップ全域を端から順に見ていけばいいだろう。ここで、壁郡探索を再帰的に行った際にチェック配列にメモを取ることによって、一度探索された壁郡は一巡の処理において再び探索されることはなくなる。

処理３．　数字伸ばし　計算量 O(XY) (正確にはもう少し小さくなる)

処理２とほぼ同じようにして処理することができるだろう。ただし、マップ全域を見て領主を探すのではなく、各領主の位置等の情報をを格納した配列を用いる。

処理４．　斜め殺し　計算量 O(XY)

処理１と同様に、簡単に実現できるであろう。

処理５．　間殺し　計算量 O(XY)

これも処理１・処理２と同様にマップ全体を見ていけばいいだろう。

処理６．　確定領土閉鎖　計算量 O(XY) (正確にはもう少し小さくなる)

各島についての、現在の領土数及び領主の位置などは、配列に格納してある。それを用いれば、領土数が必要な分に達している島は用意に知ることができる。それを用いて深さ優先探索を行い、島に隣接する空白マスを全て壁にすればよい。

処理７．　領土伸ばし　計算量 O(XY)

処理３と本質的には変わらないが、データの扱い方がやや異なるため、別にして考えた。

処理８．　領主探し　計算量 O(XY) (ただしおよそ 5*XY)

幅優先探索により、各マスに到達可能な領主が複数であるか、到達可能な領主が唯一であるならばその id を求め、配列データとして得ることができる。それを用いて、ノラ領地の領主を決定することができる。

処理９．　到達不可能マス潰し　計算量 O(XY) (ただしおよそ 5*XY)

幅優先探索により、処理８と同様に実現可能。

処理１０．　全列挙からの確定潰し　計算量 O($XY2^T$) (ただし最悪の場合。T は、処理を遂行する到達可能な空白の数についての最大値であり、任意設定である)

１つの処理としては、実装の手間・計算量ともに最も重いであろう処理である。処理の概略は以下の (1)〜(3) である。(1) １つの未完成の島に関して、その島が伸びることのできる全ての位置にある空白をメモした配列を、幅優先探索により得る。　(2) メモされた空白の位置について、その島の不足領土数分だけを取る (正確には、Combination の実装は難しいことから、取った空白の数が不足領土数に等しくない場合は棄却する) という試行を全通り試し、島が成立する場合にに関してのみ、その時の閉鎖壁も含めてそれぞれのマスが壁か領土かのデータを取る。(3) 壁になった回数が列挙の回数に等しい場合はそのマスは壁であることが確定する。領土についても同様である。

仮定的処理　計算量 O(2^A) (ただし、A は仮定の深さ)

各マス仮定をの深さ (=仮定状態) を記録しておく 36*36 の配列を用意すれば実現できる。すなわち、仮定を１つする度に、全体を統括する仮定のフェイズが 1 上がり、仮定 n の段階で処理１〜１０によって決まったマスには仮定番号 n を記録しておけばよい。仮定が棄却され、仮定を遡る際 (ここでは n ⇒

n-1 としよう) には、仮定番号 n がついたマスは元に戻し、さらに仮定元であるマス (仮定の親という) は仮定 n-1 の元に空白であるということを記録する。マスを元に戻す、と言ったがこれは、そのマスの状態 (ここでは壁、領地、ノラ領地などの情報) が変化する度に、変化前の状態を記録しておくことによって実現が可能である。

3. 実装

詳細な実装については Program2 のソースコードを参照。以下ではプログラムを読む上での注意点のみに留める。

コード上には十分な量のコメントを施してあるため、それを見ながら軽く読み進めて頂ければ、だいたいの概要は楽につかめるであろう。また、もしプログラム全体を熟読して頂ければ、プログラムの挙動を完全に理解して頂けるはずである。関数名についてもかなり直感的で、名前から機能が用意に推測できるよう考慮した。

本プログラム、書く側・読む側の分かり易さのために、複数の関数間で共有したい値については、グローバル変数を多用している。使用する度に初期化を行うことによって、書く側の保守性の危険性はほぼ無視できるだろうし、読む側にとっても、宣言の段階でその使われ方を知る必要はあまりない (つまり、その変数が使われた段階で、初めて意味を知ればいいのである)。

また、その値の意味・仕様がある程度重要になってくるマップ関連の配列については、できる限りの解説を宣言時のコメントにて行った。これについてはよく留意して頂きたい。

最後に、やや混乱しやすい部分として、SLAVE、KABE、SPACE 等のマスの状態があるので、それについて解説する。

(1) マス (x,y) が壁である。　⇔　map[x][y] == KABE
(2) マス (x,y) が空白である。　⇔　map[x][y] == SPACE
(3) マス (x,y) が数字 (領主) である。　⇔　map[x][y] == 1〜9
(4) マス (x,y) が (領主の決まった) 領土である。⇔　map[x][y] == SLAVE && master_map[x][y] != NORA
(5) マス (x,y) が、ノラ領土である。　⇔　map[x][y] == SLAVE && master_map[x][y] == NORA

4. 実行結果

・各 sample についての実行結果と所要時間 (いずれも 5 回平均)

sample4x4.pzl	0.000sec
sample5x5.pzl	0.000sec
sample10x10.pzl	0.000sec
sample20x20.pzl	-
sample24x14.pzl	0.682sec
sample36x20.pzl	1.398sec
sample36x36.pzl	-

sample20x20.pzl と sample36x36.pzl はいずれも一時間以内に終了できなかったので、計測不可能とする。

計算速度については、やはりネックになるのは仮定処理であろう。仮定の深さが大きくなれば、結局のところ、計算に要する時間は 2 の累乗で増加していくので、sample20x20.pzl と sample36x36.pzl な

どの問題は解くことができなかった。

・メモリ使用量

windows のタスクマネージャーを用いて、sample36x20.pzl を解く際のメモリ使用量を調べた所、2376KB のメモリを消費していることが判明した。このメモリ使用量は、解く問題に関わらず、概ね一定であるようだ。

この値は決して小さいとは言えないが、今回のプログラムでは、省メモリ性よりもプログラムの分かり易さ・簡便化を優先させたため、これよりもメモリ使用量をある程度小さくすることは容易に可能であると思われる。例えば、flag 管理しか行わない 36*36 の int 型配列を用意するのは、明らかなメモリ上の無駄である (int 型は 4byte の領域を持つ)。今回のプログラムでは、int 型の 36*36 配列 (int 型を構成要素に含む構造体も含む) が非常に多く使われているが、その多くは char 型で代替が可能である (char 型は 1byte の領域を持つ)。

5. データ構造

今回、幅優先探索を実現するにあたって、Queue と呼ばれるデータ構造を簡易に実装したが、これについては特に論じることもないだろう。主な機能として、データ (ここでが構造体 Masu に限定している) を加える操作と、中に入ったデータを取り出す操作を行うことができる。取り出されるデータの順番として、最初に追加されたものから取り出されるのが最大の特徴である。今回は極めて簡易に、配列及び、書き込み位置と取り出し位置を記憶した変数によって実装した。

6. 考察

人間がパズルを解く際に行うであろう大方のメソッドは、今回のプログラムにおいて再現した。

今回取り入れなかった処理として、関節点 (連結グラフにおいて、その点 (ノード) を除去した際にグラフが連結でなくなる点のことを言う) を用いる手法がある。この手法は、処理２ (壁伸ばし) を拡張する概念で、全ての壁のマスをノード、壁のマスが隣接、あるいは空白を挟んで到達可能であるすることをエッジと考えると、関節点となるノード (マス) については、必ず壁になる。これはかなり有効な手法であるだろう。関節点についての詳細なアルゴリズムを論じることはこの場では控えるが、いつか機会があれば実装したいと思う。

仮定法については、やはりマップを端から順に仮定していく手法には無理があった。仮定の深さが浅ければ、今回実装した仮定法でも現実的な時間で解を求めることは可能だが、やはり処理時間に２の累乗を要する手法は厳しく、使い物にならない場合が多い。sample10x10.pzl については、仮定法だけを用いても現実的な時間で解を求めることができた。

ちなみにだが、二コリのパズルは本来、仮定法 (try&error) を用いなくても解けるように作られている。従って、本課題の sample36x36.pzl の様に、明らかに仮定法を使わなければ解けないような問題は本来存在してはならない問題であるのだ。これらの冒涜的な悪問が解けないことは、ぬりかべパズルを解くプログラムを作る上で、何も問題とはならないと言えるだろう。

7. 感想

プログラムを書く上でつらいのはやはり、デバッグ作業だと痛感する日々であった。今後は初めからバグを生まないように精進したい。

8. 参考文献

なし。