J. Symbolic Computation (1990) **10**, 111-137

The (n^2-1)-Puzzle and Related Relocation Problems

DANIEL RATNER AND MANFRED WARMUTH

Computer and Information Sciences
University of California, Santa Cruz, CA 95064 USA

(Received October 1, 1986)
(Revised July 18, 1989)

The 8-puzzle and the 15-puzzle have been used for many years as a domain for testing heuristic search techniques. From experience it is known that these puzzles are "difficult" and therefore useful for testing search techniques. In this paper we give strong evidence that these puzzles are indeed good test problems. We extend the 8-puzzle and the 15-puzzle to an $n \times n$ board and show that finding a shortest solution for the extended puzzle is NP-hard and is thus believed to be computationally infeasible.

We also sketch an approximation algorithm for transforming boards that is guaranteed to use no more than a constant times the minimum number of moves, where the constant is independent of the given boards and their side length n.

The studied puzzles are instances of planar relocation problems where the reachability question is polynomial but efficient relocation is NP-hard. Such problems are natural robotics problems: A robot needs to efficiently relocate packages in the plane. Our research encourages the study of polynomial approximation algorithms for related robotics problems.

1. Introduction

For over two decades the 8-puzzle and the 15-puzzle of Sam Loyd [1959] have been a laboratory for testing search methods. Doran and Michie used these games in their general problem-solving program, called Graph Traverser [Doran & Michie, 1966]. Pohl used the 15-puzzle in his research on bi-directional search and dynamic weighting [Pohl, 1971; Pohl, 1973; Pohl, 1977]. Gaschnig described the 8-puzzle as "...simple, convenient to manipulate, yet exhibiting interesting phenomena that hypothetically hold for a broader class of subjects..." [Gaschnig, 1979]. Recently Korf used these puzzles as examples for the method of solving search problems by building Macro-Operators [Korf, 1985a] and as examples for *IDA** search algorithm [Korf, 1985b]. Judea Pearl used the 8-puzzle throughout the first half of his book on heuristics as one of the main examples [Pearl, 1984]. Also, in the last decade, when learning algorithms have started to blossom, some of the empirical tests of these algorithms were performed on the 8-puzzle and the 15-puzzle [Politowski, 1986; Rendell, 1983].

The main reasons for selecting these problems as workbench models for measuring the performance of searching methods are:

0747-7171/90/080111+27 $03.00/0

1) There is no known algorithm that finds a shortest solution for these problems efficiently.
2) The problems are simple and easy to manipulate.
3) The problems are good representatives for a class of problems with the goal of finding a relatively short path between two given vertices in an undirected graph.
4) The size of the search graph is exponential in the length n of the side of the board, even though the input configurations can be described easily by assigning a location for each tile (there are n^2 tiles).
5) The search graph can be specified by a few simple rules.

Certainly, if there existed simple efficient algorithms for finding a shortest solution for these problems, then heuristic approaches would become superfluous. Thus we need to give a convincing argument that no such algorithm exists. This is accomplished by using complexity theory an approach suggested by Goldberg and Pohl [1984]. We show that finding the shortest solution for a natural extension of the 8-puzzle and the 15-puzzle is NP-hard. Thus unless P=NP, which is considered to be extremely unlikely, there is no polynomial algorithms for finding a shortest solution. Of course, since the number of distinct configurations in the 8-puzzle and the 15-puzzle is finite, theoretically (and practically for the 8-puzzle) one can find shortest solutions for all the possible inputs by analyzing the whole search graph. To get problems of unbounded size we extend the problem to the $n \times n$ checker board and call the puzzle of that board size the (n^2-1)-*puzzle*. (The tiles numbered one through n^2-1 occupy all but one square which is empty, that is it contains the *blank tile*.) The 8-puzzle and the 15-puzzle are the (n^2-1)–puzzle , where n is 3 and 4, respectively. The same natural extension to the $n \times n$ board was used by Lichtenstein and Sipser for analyzing the complexity of Go [Lichtenstein & Sipser, 1978] and by Fraenkel et al. for analyzing the complexity of Checkers [Fraenkel et al., 1978].

The aim of the (n^2-1)–puzzle is to find a sequence of moves which will transfer a given initial configuration of an $n \times n$ board to a final (standard) configuration. A *move* consists of sliding a *tile* onto the empty square from an orthogonally adjacent square. We will show that the following decision problem, called *nPUZ*, is NP-complete:

INSTANCE. two $n \times n$ board configurations and a bound k.
QUESTION. is there a solution for transforming the first (initial) configuration into the second (final) configuration requiring at most k moves?

The pebble games of Kornhauser et al. [Korhnhauser, Miller & Spirakis, 1984] can be viewed as a direct generalization of the (n^2-1)–puzzle : Let G be a graph with n vertices and $b < n$ pebbles numbered $1, \cdots, b$. The distinct pebbles (which correspond to the non-blank tiles of the (n^2-1)–puzzle) must always be on different vertices. A *move* consists of sliding a pebble along an edge of the graph to an adjacent unoccupied vertex. They address the question of reachability, i.e. whether a final configuration of the pebbles is reachable from an initial configuration using any legal sequence of moves. It was shown that the general reachability problem can be decided in polynomial time. In the case of (n^2-1)–puzzle the graph is of a very restricted form (the $n \times n$ planar grid) and the number of pebbles (non-blank tiles) b is one less than the number of vertices. Note that for the (n^2-1)–puzzle reachability is easy to decide as well.

In this paper we are concerned with the complexity of reaching the final configuration from the initial configuration in a small number of moves. The following version of the decision problem *nPUZ* for general graphs was considered by Goldreich [Goldreich, 1984]:

INSTANCE. a graph of n vertices and two configurations of the pebbles $1, \cdots, n-1$ on the graph and a bound k.
QUESTION. is there a solution for transforming the first configuration into the second configuration requiring at most k moves?

Goldreich gave a very elegant reduction for showing that the above decision problem is NP-complete [Goldreich, 1984]. In the first part of the paper we show that *nPUZ* is NP-complete as well. Note that now the graphs are of a very restricted form. For each length of the side of the board there is only one square-shaped grid graph and this graph is very symmetric. Intuitively we cannot encode any "hardness" into the structure of such graphs and is not surprising the NP-completeness reduction given in this paper is much more involved than the Goldreich's reduction for general graphs.

In the puzzle tiles need to be relocated on the planar grid. The relocation task, even without the specific rigid rules of the game, is the essence of the NP-hardness of *nPUZ*. We consider a *relocation procedure* on graphs which is different than the one considered by Kornhauser et al. [Kornhauser et al., 1984] and is again related to (n^2-1)–puzzle. In this relocation procedure *elements* that reside in vertices of a graph are shipped to other vertices. The shipping process is done step by step along a path in the graph. Assume that the path consists of the vertices $(v_0, v_1, \cdots, v_t)$. If at the beginning of step i the procedure is at v_{i-1} in which there are some elements, then at the end of this step (the beginning of the next step) the procedure is at v_i. When the procedure moves along the edge (v_{i-1},v_i) it may ship some of the elements from v_{i-1} into v_i. We first show the intractability of a relocation problem on planar graphs. This problem, called the *REL* problem, is less restrictive and easier to prove NP-complete:

INSTANCE. A planar directed graph $G(V,E)$ where each $e \in E$ has capacity zero or two, a set X of elements, and an initial and final configuration. A *configuration* specifies the location of each element of X at the vertices of V.
QUESTION. Is there a relocation procedure that ships the elements of X from their initial configuration to their final configuration such that the procedure moves along each $e \in E$ exactly once (the shipment is along an Eulerian path) and it never ships along each edge more elements than allowed by the capacity of the edge?

The *nPUZ* and *REL* problems can be viewed as robotics problems: A robot needs to efficiently relocate objects in the plane. In the case of *nPUZ* the objects have a simple shape. The relocation of arbitrarily shaped objects through areas with arbitrary boundaries is of much higher complexity. Hopcroft et al. have shown that in that case the question of reachability is already PSPACE-complete [Hopcroft, Schwartz & Sharir, 1984].

The NP-completeness proof of *nPUZ* will simulate the simpler proof for *REL*. The graph is mapped onto the board of the puzzle problem. The vertices and edges will correspond to certain areas of the board. The elements and the capacities are encoded by the arrangements of tiles in these areas in the start and final configuration.

Since finding the shortest solution is NP-hard we would like to know how close the shortest solution can be approximated in polynomial time. In the second part of the paper we show that finding a solution that is any fixed additive constant away from the optimum is also NP-hard. However we have positive results for approximating the optimum by a multiplicative constant. In this paper we are only concerned with proving that such a constant exists. The given proof would lead to a rather high constant. We discuss how the constant may be optimized. It is an open problem to find the polynomial approximation algorithm with the best

possible constant. Note that this type of algorithm finds reasonable solutions even if n is large (around 100) and there are no pathological cases as there are for search methods. The research based on search methods only addresses the cases of n=3,4 and no bounds are given as to how the length of the heuristic solution is related to the length of the optimum solution.

As suggested by Ratner and Pohl [Ratner, 1986; Ratner & Pohl, 1986], we recommend a combined approach for solving search problems: Use an approximation algorithm whose solution has a provable worst case bound and then do local optimization of the approximate solution, employing AI search methods.

This paper is outlined as follows. In Section 2 we describe a special symmetric version of the satisfiability problem, called 2/2/4–*SAT*, which is shown to be NP-complete in the Appendix. Section 2 also contains the basic definitions and properties of (n^2-1)–puzzle used throughout the paper. In Section 3 we reduce 2/2/4–*SAT* to *REL* and in Section 4 we give a similar reduction for *nPUZ*. In the last section we describe the approximation algorithm for *nPUZ*.

We suggest to apply our approach to other puzzles, like the Rubik's cube, which are also used extensively as examples in the AI-literature. Recently, search methods have been devised for finding the shortest solution for the $n \times n \times n$ Rubik's cube, which are computationally feasible for the case of n=2 and n=3 [Fiat et al., 1989]. Is the problem of finding a shortest solution for the $n \times n \times n$ Rubik's cube NP-hard? Are there polynomial time approximation algorithms for $n \times n \times n$ Rubik's cube that approximate the optimal solution by a multiplicative constant?

Is there an approximation algorithm (off by only a multiplicative constant) for the pebble game on general graphs considered by Kornhauser et al. [Korhnhauser, Miller & Spirakis, 1984]? As for more general relocation problems we would like to know whether there are approximation algorithms (off by a multiplicative constant) for the following setting: Given a planer graph where packages reside at the vertices (cities), and a truck that can carry at most c packages. Each package has an initial and a final city. The aim is to relocate the packages with the truck using the shortest possible route. Approximation algorithms have been found for special cases of this problem: the Traveling Salesman Problem and the Vehicle Routing Problem (with one vehicle) [Lawler et al., 1985].

2. Basic Definitions

The classical NP-complete problem is the satisfiability problem: given a boolean formula in conjunctive normal form, does there exist a truth setting that satisfies the formula. We use a simple symmetric version, called 2/2/4–*SAT*, in the reductions for *REL* and *nPUZ*. This version is shown to be NP-complete in the Appendix.

2/2/4–*SAT*:
INSTANCE. A set U of m variables and a collection C of m clauses over U such that each clause $c \in C$ has exactly four literals ($| c | = 4$) and each $u \in U$ appears in all clauses of C exactly 4 times, twice negated and twice unnegated.
QUESTION. Is there a truth assignment for U such that each clause in C has exactly two true literals?

In the rest of this section we give the main definitions and basic facts of (n^2-1)–puzzle that are needed throughout the paper. The puzzle is played on an $n \times n$ checker board. There are n^2 distinct *tiles* on the board; one *blank tile* and n^2-1 tiles numbered from 1 to n^2-1.

Each of the n^2 square *location* of the board is occupied by exactly one tile. An instance of (n^2-1)–puzzle consists of two board configurations B_1 (the *initial configuration*) and B_2 (the *final configuration*). A *move* is an exchange of the blank tile with a tile located on an horizontally or vertically adjacent location. The goal of (n^2-1)–puzzle is to find a short sequence of moves that transforms B_1 to B_2. *nPUZ* is the corresponding decision problem. The question is whether, for given B_1, B_2 and k, the configurations can be transformed into each other in at most k moves.

Given a sequence that solves an instance of *nPUZ* by moving B_1 to B_2 in at most k moves, then the same sequence solves any instances in which the tiles (other than the blank tile) of both configurations are renumbered in the same way. Thus the set of possible solutions for the instance of *nPUZ* is determined by the relative location of the tiles in B_1 and B_2.

Not every two configurations are reachable from each other. The configurations can be described as a permutation: For any tile t let $\pi(t)$ be the tile that resides in the same location in B_2 as the tile t in B_1. This permutation can be decomposed into cycles, where a cycle is a sequence of tiles $(t_1, \cdots, t_q)$ s.t. t_i appears in B_1 on the same location as $t_{(i+1) \bmod q}$ in B_2. A cycle is called *even* if its size is even. Reachability can be characterized using the parity of the cycles.

THEOREM 1. *Let B_1 and B_2 be configuration in which the blank tile is at the same location. Then B_1 and B_2 are reachable iff their permutation has an even number of even cycles.*

A proof of this theorem can be found in [McCoy & Berger, 1972]. It is easy to construct the permutations and check reachability in $O(n^2)$ time. The above theorem implies that if two configurations that are the same except for two non-blank tiles which are swapped, then the configurations are not reachable. The following fact is easily checked and left as an exercise:

FACT 1. There is an algorithms that transforms any two reachable configurations into each other, never using more than $p(n)=q\,n^3$ moves, where q is a constant independent of the size of the board n. The running time to the algorithms is proportional to $p(n)$.

Let l_1 and l_2 be two locations on the board and assume the blank tile is to be advanced from l_1 to l_2 in the minimum number of moves. Consider a rectangle of locations s.t. l_1 and l_2 are in opposite corners. One way to achieve the minimum number of moves is to move from l_1 along one of the two *orthogonal angles* of the rectangle. The (Manhattan) *distance* between the two locations, denoted by $d(l_1,l_2)$, is the minimum number of moves required to move the blank tile from one location to the other. Note that this notion of distance is different from the Euclidean distance.

The distance of a tile x in two configurations B_1 and B_2 is the distance between the two locations of x on the board and the distance between two configurations B_1 and B_2, denoted by $d(B_1,B_2)$, is the total distance of all non-blank tiles.

Let S be a sequence of moves that transforms B_1 to B_2. Using the notion of distance we can partition the moves as follows. Assume we are to move a non-blank tile x. Let *min* be the smallest distance of x from its final location in B_2 in all previous and current configurations. If after the move the distance of x to its location in B_2 is *min*-1, then this move is a *necessary move*, otherwise it is an *unnecessary move*. Clearly the total number of necessary moves is equal to $d(B_1,B_2)$ and this is a lower bound on the number of moves required to transform B_1 to B_2.

3. Relocation in a Graph Via an Eulerian Path

In this section we prove that relocating elements that reside in vertices of a planar graph via an Eulerian path is NP-complete (the problem *REL*). We present *REL* in this paper because the NP-completeness reduction from *2/2/4–SAT* to *REL* is very similar to the reduction to our goal problem *nPUZ*. In *nPUZ* tiles must be relocated following the rules of the puzzle. The relocation problem for planar graphs captures the hard core of *nPUZ*. It allows us to ignore the rules of the puzzle for a moment.

THEOREM 2. *REL is NP-complete.*

PROOF. Let $U=\{u_1, u_2, \cdots, u_m\}$ be a set of variables and $C=\{c_1, c_2, \cdots, c_m\}$ be a set of clauses defining an arbitrary instance of *2/2/4–SAT*. From this instance we will construct an instance of *REL*. Such an instance is a graph $G(V,E)$ with capacities (zero or two) for each $e \in E$, a set X of elements, an initial configuration (called B_1), and a final configuration (called B_2). First we start with the description of the graph and later we define the configurations.

The graph (Figure 2) consists of $5m+2$ vertices and $12m-3$ edges. The vertices are divided to four groups. The first group is built up from m *diamonds* of four vertices and four edges of capacity 2 each. The i-th diamond which is shown in Figure 1 corresponds to the variable u_i. This diamond contains the vertices: top_i, nu_i, bot_i, and $\overline{nu}_i$.

The second group is the single vertex TC (stands for truth collection). The third group is the single vertex FC (stands for false collection). The fourth group consists of m vertices. The i-th vertex of this group, called nc_i, corresponds to the i-th clause in the boolean formula of the *2/2/4–SAT* instance.

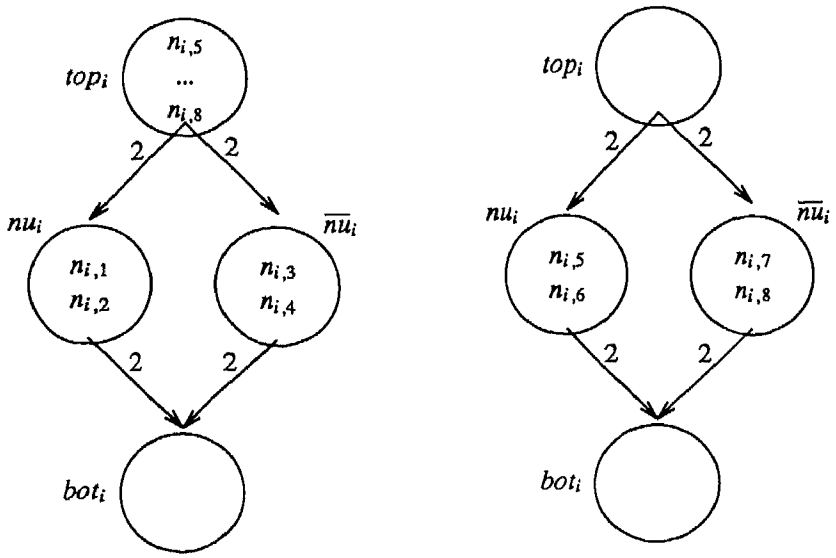

Figure 1: The i-th diamond in B_1 and B_2.

The directed edges connecting the vertices and the capacities of the edges are specified in Figure 2. Two edges of capacity 2 are outgoing from each diamond, one to TC and one to FC. Two edges of capacity zero are ingoing to each diamond (except for the first one), one from TC and one from FC. There is only one ingoing edge ((nc_m, top_1)) to the first diamond.

Similarly, two edges of capacity 2 are directed into each clause vertex nc_i, one from TC and one from FC. Two edges of capacity zero are leaving from each clause vertex (except for nc_m), one to TC and one to FC. Only one edge ((nc_m, top_1)) of capacity zero is leaving the last clause vertex (nc_m).

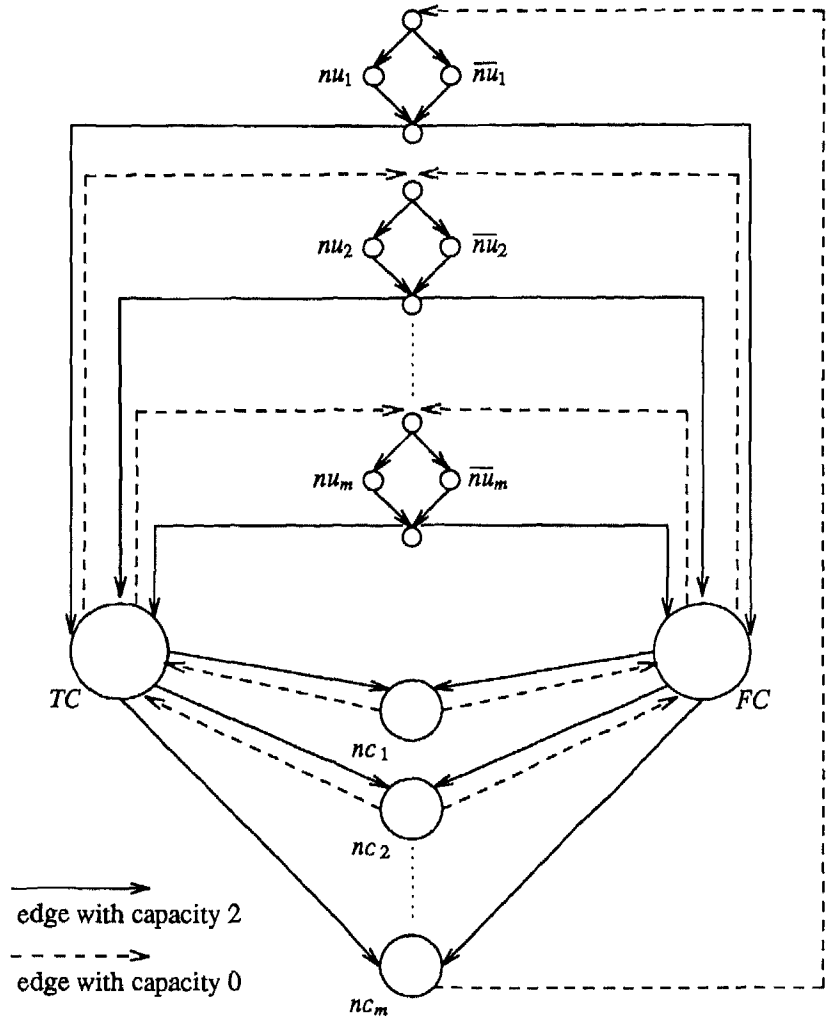

Figure 2: The graph of the *REL* instance.

To complete the definition of the instance of *REL* we need to specify the set elements X and the initial and final locations of the elements in the graph. The set X consists of $8m$ elements. Recall that in 2/2/4–*SAT* each variable occurs twice negated and twice unnegated. There is an element for each of the four occurrences and these elements are called *literal* elements: $n_{i,1}$ and $n_{i,2}$ correspond to the two appearances of u_i in C, and $n_{i,3}$ and $n_{i,4}$ correspond to the two appearances of $\overline{u}_i$ in C. The remaining $4m$ elements $n_{i,5}$, $n_{i,6}$, $n_{i,7}$ and $n_{i,8}$ are called *filler* elements. The fillers are required for technical reasons in the *nPUZ* reduction. For this reduction they are not needed. However, we add them to the instance of *REL* make both reductions similar.

In B_1 the literal elements are located in the diamonds as specified in Figure 1, the pair $n_{i,1}$, $n_{i,2}$ is in nu_i and the pair $n_{i,3}$, $n_{i,4}$ is in $\overline{nu}_i$. In B_2 these elements are in the vertices that correspond to the clauses. The four elements that are associated with the four literals of the i-th clause appear in vertex nc_i. In B_1 the fillers $n_{i,5}$, $n_{i,6}$, $n_{i,7}$ and $n_{i,8}$ are located in top_i. In B_2, $n_{i,5}$, $n_{i,6}$ are located in nu_i and $n_{i,7}$, $n_{i,8}$ are located in $\overline{nu}_i$. All the remaining vertices contain no elements. This completes the definition of the instance of *REL* and the following two claims complete the proof of the theorem.

CLAIM 1. If there is a truth assignment $f: U \longrightarrow \{T, F\}$ that satisfies the 2/2/4–*SAT* instance then there is a relocation procedure along an Eulerian path that shifts B_1 to B_2.

PROOF. The proof is constructive. First we ship all the $n_{i,j}$ elements that correspond to true literals from their vertices in B_1 to the TC vertex. This collection is done by the following procedure.

PROCEDURE SHIFT 1.
for $i := 1$ to m do begin
    if $f(u_i) = T$ then
        move along (top_i, nu_i) with $n_{i,5}$ and $n_{i,6}$;
        move along (nu_i, bot_i) and (bot_i, TC) with $n_{i,1}$ and $n_{i,2}$;
    else
        move along $(top_i, \overline{nu}_i)$ with $n_{i,7}$ and $n_{i,8}$;
        move along $(\overline{nu}_i, bot_i)$ and (bot_i, TC) with $n_{i,3}$ and $n_{i,4}$;
    if $i \neq m$ then move along (TC, top_{i+1}) ;
end.

When the above loop is finished, then the vertex TC contains $2m$ literal elements. Each diamond contributes exactly two elements. The two elements from the i-th diamond are either $n_{i,1}$ and $n_{i,2}$ (from nu_i) or $n_{i,3}$ and $n_{i,4}$ (from $\overline{nu}_i$). The two literal elements that are in TC were replaced by the fillers $n_{i,5}$ and $n_{i,6}$ or by $n_{i,7}$ and $n_{i,8}$ from top_i.

The next step drops the $2m$ literal elements that are in TC into the nc_j vertices they belonged to in B_2. As mentioned above, these $2m$ elements correspond to the $2m$ *true* literals. Since the truth assignment f solves the 2/2/4–*SAT* instance, it follows that two literal elements from each clause vertex nc_j in B_2 are now in TC. These elements are dropped into their clause vertices by the following Procedure Shift 2:

PROCEDURE SHIFT 2.
for j:=1 to m do begin
    move along (TC, nc_j) with the two literal elements that
    are in nc_j in B_2;
    if $j \neq m$ then move along (nc_j, TC) ;
end.

As a result of the above procedure, each nc_j vertex receives the two elements that correspond to the true literals of the i-th clause in the 2/2/4–*SAT* instance.

Now we move along (nc_m, top_1). From this point we do a second pass over the diamonds and the clause vertices repeating the two procedures Shift 1 and Shift 2 given above with slight modifications. In Shift 3 we collect all the $n_{i,j}$ elements that correspond to false literals into the FC vertex. This is done by traversing all the edges of the diamonds that have not been traversed in the first pass and by traversing all the edges that connect FC with the diamonds.

After Shift 3 the $2m$ elements that correspond to the $2m$ false literals are in FC. In the final procedure Shift 4 the algorithm drops the $2m$ literal elements from FC into the nc_j vertices in which they appear in B_2. Once the second pass is completed the arrangement of all elements in the graph is as prescribed in B_2. Observe that each edge is traversed exactly once and the number of elements moved through each edge always equals the capacity of the edge.

CLAIM 2. If there is a relocation procedure that ships the elements from B_1 to B_2 along an Eulerian path then there is a truth assignment $f: U \longrightarrow \{T,F\}$ that satisfies the 2/2/4–*SAT* instance.

PROOF. We need to ship the four literal elements from their initial locations in the i-th diamond to the clause vertices (the nc_j) they belong to in B_2. These literal elements must pass through bot_i. There are only two edges (bot_i,TC) and (bot_i,FC) outgoing from bot_i. Both edges have capacity two. This means that when the procedure moves along (bot_i,TC) and (bot_i,FC) it must carry two literal elements each time. Furthermore, the first time the procedure ships two literals to bot_i they must be either the pair $(n_{i,1},n_{i,2})$ or the pair $(n_{i,3},n_{i,4})$. Now the procedure must continue to move along with these two literals. Thus, for each i, $1 \le i \le m$, the given procedure ships the pair $(n_{i,1},n_{i,2})$ along (bot_i,TC) or along (bot_i,FC).

Let us define the truth assignment $f: U \longrightarrow \{T,F\}$ as follows:

$f(u_i) = T$ if the procedure ships the pair $(n_{i,1},n_{i,2})$ along (bot_i,TC).

$f(u_i) = F$ if the procedure ships the pair $(n_{i,1},n_{i,2})$ along (bot_i,FC).

Note that if $f(u_i) = T$ (respectively F) then the procedure ships the pair $(n_{i,3},n_{i,4})$ along (bot_i,FC) (respectively (bot_i,TC)).

We proceed to show that the above truth assignment satisfies the requirements of the 2/2/4–*SAT* instance. There are two ingoing edges to each nc_j vertex, each of capacity two. There is no way to ship elements from TC to FC or vise versa (see Figure 2). Thus the procedures ships from TC exactly two literal elements to each of the nc_j vertices. According to the definition of f these elements correspond to true literals. The other two elements that arrive at each nc_j vertices are from FC, which means that they correspond to false literals. This completes the proof of Claim 2 and Theorem 1. □

In the specification of *REL* we assumed that the capacities of the edges are either zero or two and that the graph is directed. It is easy to show that *REL* remains NP-complete if the graph is directed and all capacities are two: For each directed edge of capacity zero we locate two dummy elements at the beginning of the edge in B_1 and these dummies appear at the end of the edge in B_2. Also notice that in the reduction of the above theorem the correct direction of the edges is implied by the relocation of the elements via an Eulerian path. Thus the undirected versions of *REL* are also NP-hard. It is an open problem whether there is a polynomial algorithm for *REL* if we only allow capacities zero or one.

4. The Intractability of *nPUZ*

In this section we give a reduction of the 2/2/4–*SAT* problem to the *nPUZ* problem that mimics the reduction given in the previous section.

THEOREM 3. *nPUZ is NP-complete.*

PROOF. The proof is divided into three subsections. In the first subsection we describe the used instance of *nPUZ* and in the last two subsections we give the two directions of the reduction.

4.1. The Instance of *nPUZ*

The instance of *nPUZ* that is constructed from a given instance of 2/2/4–*SAT* is similar to the instance of *REL* used in the previous section. We will map the graph of Figure 2 onto the board. The instance of *nPUZ* consists of two $n \times n$ board configurations B_1 (the initial configuration), B_2 (the final configuration) and an integer k which is an upper bound on the number of moves that can be used to transform B_1 to B_2. To simulate the graph of Figure 2 we have to capture the notions of vertices, edges, elements, relocation, moving along an edge and capacity of an edge. Each vertex in the graph of Figure 2 corresponds to a square area of locations (we call these areas *nodes*). Edges are identified as short sequences of thin horizontal and vertical stripes of locations that connect the nodes of the edge. Figure 3 gives a rough outline of how the graph of *REL* (Figure 2) is mapped onto the board. It shows all the locations at which B_1 and B_2 might differ. The main difference is a 45 degree counterclockwise rotation. The bold line segments are stripes of locations on the board of width three. These stripes belong to edges of capacity two. Similarly, the dashed lines are stripes of width one and belong to edges of capacity zero. All square areas (the nodes) are three locations wide except for areas of *FC* and *TC* whose length of their side is $4m$. The latter two nodes are called the *large nodes* and the remaining ones the *small nodes*. All pieces of Figure 3 (the stripes of the edges and the nodes) are composed of disjoint locations.

Recall that we want to describe the two board configurations B_1 and B_2 of the instance of *nPUZ*. Figure 3 shows all the locations at which B_1 and B_2 might differ: in the stripes of the edges and the nodes. That is, every tile that lies outside the stripes and the nodes is in same location in B_1 and B_2.

To complete the specification of Figure 3 we still need to give the lengths of the stripes. Each stripe is required to be at least $5l$. The basic length unit l is defined in terms of u which will be shown to be an upper bound on the number of unnecessary moves used in the transformation of B_1 to B_2 from a given truth assignment: $l = u+16m+1$ and $u = 508m + 29m\,p(6) + 9m\,p(4m+3)$ (p is the polynomial given in Fact 1 of the previous section). It is easy to assign explicit locations to the all nodes and stripes of the edges such that the above specifications (summarized in Figure 3) are followed. This can be done s.t. the length of the longest stripe and the distance between any two nodes is at least $5l$ and the length n of the side of the whole board is $O(ml)$.

We now address how the 2/2/4–*SAT* instance is encoded in the two board configurations B_1 and B_2. Let $U=\{u_1, u_2, \cdots, u_m\}$ be a set of variables and $C=\{c_1, c_2, \cdots, c_m\}$ be a set of clauses defining an arbitrary instance of 2/2/4–*SAT*. Similar to the *REL* instance of the previous reduction, the two occurrences of u_i correspond to the elements (here explicit tiles) $n_{i,1}$ and $n_{i,2}$ located in (the area of) nu_i of B_1. The two occurrences of $\overline{u}_i$ are responsible for the element tiles $n_{i,3}$ and $n_{i,4}$ in $\overline{nu}_i$ of B_1. Again as in the instance of *REL*, these $4m$ element tiles (called literal tiles) are distributed among the nodes nc_i in B_2. The area of nc_i receives the four literal tiles corresponding to the four literals of c_i. Analogously, there are four fillers tiles $n_{i,5}$ to $n_{i,8}$ that are located in the node top_i in B_1. In the final configuration B_2 the filler tiles $n_{i,5}$ and $n_{i,6}$ appear in nu_i and $n_{i,7}$ and $n_{i,8}$ appear in $\overline{nu}_i$.

We next describe the relative location of the tiles of the edges in B_1 and B_2. Recall that the edges of capacity zero (*zero–edges*) are composed of stripes of width one. For any zero-edge in B_1 simply move all its tiles one step backward w.r.t. the direction of the edge (see Figure 4). Observe that one tile leaves the edge at the tail of the edge and another tile is drawn into

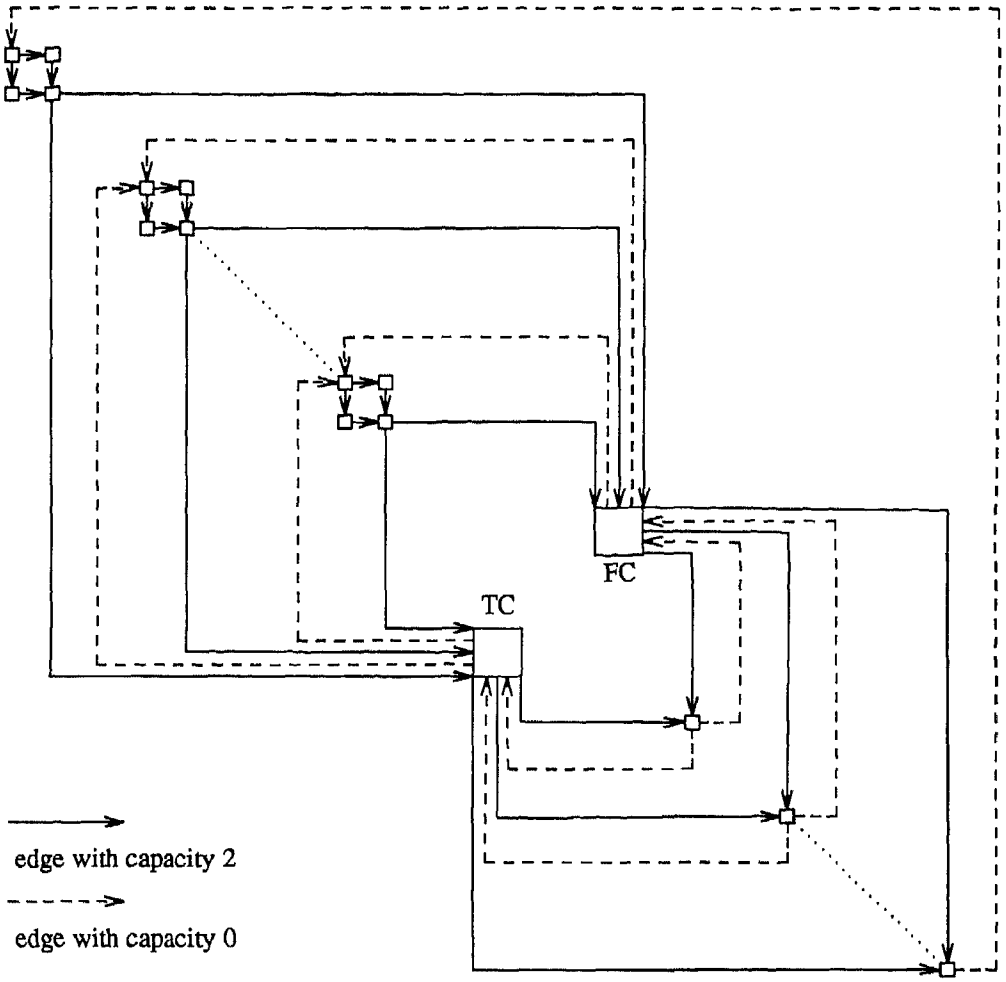

Figure 3: The locations in which B_1 and B_2 might differ.

the edge at the head of the edge. For the latter tile select any tile from the node that is adjacent to the head which is not an element tile.

The relative location in B_1 and B_2 of the *two-edges* is more involved. We relocate the tiles of B_1 as prescribed in Figure 5 (The figure shows a right turn of a two-edge). Note that each tile (except for some tile in the corners of the edges) moves one step backward w.r.t. the direction of the edge and some tiles also move one step sideways. Furthermore, three tiles move off the tail of the edge and the same number is drawn into the edge at the head. For the latter three tiles again select any three non-element tiles from the node adjacent to the head of the edge.

It is easy to verify that during the construction of B_2 the number of tiles that leave and enter each node is the same. For example, six tiles leave nc_m via the two adjacent heads of two-edges and one tile is drawn into nc_m by the tail of the adjacent zero-edge. The imbalance of five tiles is compensated by the fact that nc_m receives the blank tile in B_2 together with four element tiles (which are in the diamonds in B_1).

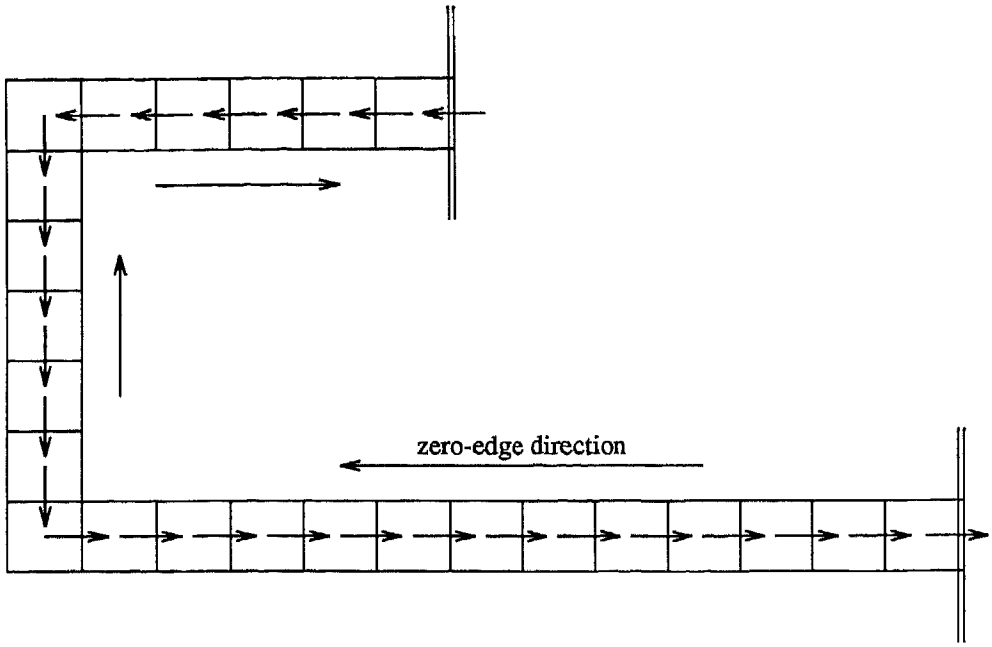

Figure 4: The relative location of the tiles of a zero-edge.

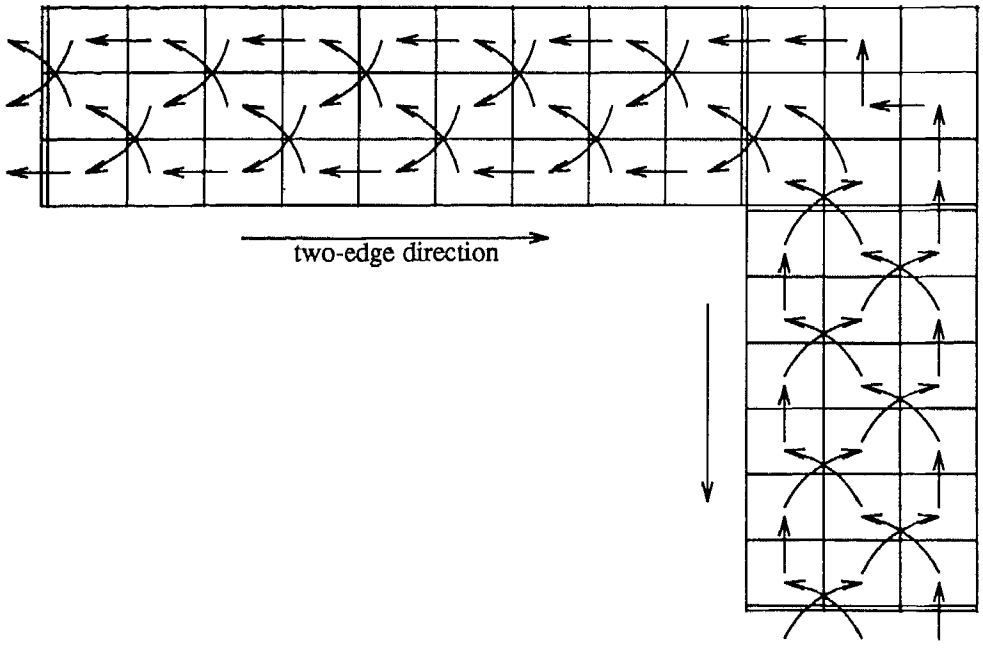

Figure 5: The relative location of the tiles of a two-edge.

We have specified the exact relative location of the tiles of B_2 w.r.t. B_1 except for where the tiles of the nodes of B_2 are located within the area of each node. Since the sizes of the nodes is negligibly small (dominated by our choice of l as shown later) the exact location of these tiles within the nodes of B_2 is not important and can be fixed arbitrarily. However, now it might be possible that B_2 is not reachable from B_1 (see Theorem 1). If so then we simply exchange two arbitrary tiles of nc_m in B_2. Now the reachability of B_2 from B_1 is guaranteed. Note that reachability between two configurations can be checked in time $O(n^2)$ (use Theorem 1).

To complete the definition of the instance of *nPUZ* we set k to $d(B_1,B_2)+u$. It is easy to see that both boards (and k) can be constructed in time polynomial in m. Recall that the length of side of the board is $O(m^5)$. In the next two subsections we show that the instance of 2/2/4–*SAT* has a solution iff B_1 can be moved to B_2 in at most k moves.

4.2. Constructing a Solution to *nPUS* from a truth setting of 2/2/4–*SAT*

For the first direction we assume that we are given a truth assignment f of the variables of U that satisfies the 2/2/4–*SAT* instance. We will move the board configuration B_1 to B_2 using a modification of the four procedures Shift used in the reduction of *REL*. The main modification is that moving along an edge is more involved in *nPUZ* than in *REL*. In *nPUZ* the empty tile moves along the stripes of the edge (possibly "carrying along" two element tiles) and while doing so rearranges the tiles in the stripes. For the case of a zero-edge the rearrangement is rather simple since no element tiles are carried along: rearrange the node at the tail of the edge s.t. the blank tile is at the location within the node that is adjacent to the tail of the zero-edge; then shift the tiles of the edge one step backwards as the empty tile traverses forwardly along the stripes of the edge (see Figure 4).

For the two-edges the rearrangement of the tiles in the stripes is much more involved. After the two element tiles and the blank tile have been moved through the two-edge the rearrangement will be as described in Figure 8. This rearrangement is achieved in two steps. First, rearrange the node adjacent to the tail of the edge such that the two element tiles x and y and the empty tile are positioned (within the node) in a straight consecutive chain and the empty tile is adjacent to the tail of the edge. See Figure 6 for details: the double bars denote the borders to the nodes (one at the head and one at the tail of the stripe).

α_{-2}	α_{-1}	α_0	α_1	α_2	α_3	α_4	α_5	....	α_{r-1}	α_r	α_{r+1}	α_{r+2}	α_{r+3}
x	y		β_1	β_2	β_3	β_4	β_5	....	β_{r-1}	β_r	β_{r+1}	β_{r+2}	β_{r+3}
γ_{-2}	γ_{-1}	γ_0	γ_1	γ_2	γ_3	γ_4	γ_5	....	γ_{r-1}	γ_r	γ_{r+1}	γ_{r+2}	γ_{r+3}

Figure 6: The arrangement of a straight two-edge in B_1.

In the second step the chain [x, y, blank tile] is moved through the edge by alternately applying the move sequences S_u and S_d to the blank tile: S_u := <*left*,*left*,*up*,*right*,*right*,*right*,*down*>; S_d equals S_u but *up* is exchanged with *down* and vice versa. Each of the seven-step sequences advances the chain be one step to the right. Figure 7 is produced from Figure 6 by applying S_u, S_d, and again S_u.

α_{-1}	α_0	β_1	α_2	α_3	β_3	α_4	α_5	....	α_{r-1}	α_r	α_{r+1}	α_{r+2}	α_{r+3}
α_{-2}	γ_{-1}	α_1	x	y		β_4	β_5	....	β_{r-1}	β_r	β_{r+1}	β_{r+2}	β_{r+3}
γ_{-2}	γ_0	γ_1	γ_2	β_2	γ_3	γ_4	γ_5	....	γ_{r-1}	γ_r	γ_{r+1}	γ_{r+2}	γ_{r+3}

Figure 7: The arrangement after advancing by three steps.

If we apply the seven-step sequences alternatingly $r+3$ times we end up with the arrangement of Figure 8 (In the figure r is odd).

α_{-1}	α_0	β_1	α_2	β_3	α_4	β_5	α_6	....	β_r	α_{r+1}	α_{r+2}	β_{r+2}	α_{r+3}
α_{-2}	γ_{-1}	α_1	γ_2	α_3	γ_4	α_5	γ_6	....	α_r	γ_{r+1}	x	y	
γ_{-2}	γ_0	γ_1	β_2	γ_3	β_4	γ_5	β_6	....	γ_r	β_{r+1}	γ_{r+2}	γ_{r+3}	β_{r+3}

Figure 8: The arrangement of a straight two-edge in B_2.

Note that the relative location of the tiles of the edge between Figure 6 and 8 is given by (the horizontal part of) Figure 5. Thus moving along the edge as prescribed moves its tiles from their location in B_1 to their location in B_2.

So far we have only shown how to move the empty tile together with two element tiles along a single stripe of capacity two (see figures 6 to 8). A two-edge might be composed of two stripes. The following sequence of moves results in a 90 degrees clockwise turn of the chain [x, y, blank tile]: $<left, left, down, right, right, up, left, down, down, right>$. This sequence produces the relative rearrangement of the corner given in Figure 5. After the rotation the chain is moved along the second stripe by alternating two rotated versions of S_u and S_d.

We still have to take care of some subtle details concerning the final arrangement of the edges and nodes. After an edge of capacity c (c=0 of c=2) is traversed, $c+1$ tiles are drawn one location across the border of the node into the head of the edge (see figures 4 and 5). At the completion of the traversal we first pull any $c+1$ tiles across the border. If the last $c+1$ locations of the tail are not arranged as in B_2, then we temporarily enlarge the node size by one location in each direction so that the $c+1$ locations are included in the enlarged node area and then rearrange the tail as prescribed in B_2.

Similarly, the last time the empty tile leaves a node during the four procedures Shift the arrangement of the node has to be as in B_2. The nodes are completed in the following order: one of $\{nu_i, \overline{nu}_i\}$ ($1 \le i \le m$), TC, the remaining nodes in the diamonds, the nc_j ($1 \le j < m$), FC, nc_m. Observe that the last exit from each node except the nc_i nodes is via a two-edge. This fact will allow us to leave each of these nodes as given in B_2 (We used filler tiles to assure that the last exit from the top_i is via a two-edge).

The last exit from a node via a two-edge is done as follows. Begin moving the chain [x, y, blank tile] outside of the node along the last outgoing two-edge but stop this process as soon as the chain has left the node (see Figure 7). Consider an *enlarged node* whose length of its side is increased by three that includes the node and the chain. Rearrange the enlarged node such that its area other than the beginning of the edge is as in B_2 and the beginning of the edge is the same as before this final rearrangement except for a possible swap of x and y in the chain. Finally continue moving the chain along the two-edge as before. Observe the swap of x and y might be necessary (see Theorem 1) to assure reachability between the two

configurations of the enlarged node. The rearrangement caused by moving [x, y, blank tile] through a two-edge is the same for all choices of x and y (in particular for [y, x, blank tile]).

We next show how the final arrangement of the nc_j nodes ($1 \leq j < m$) can be achieved. The last entry into a nc_j node is via a two-edge from FC and the last exit via a zero-edge. Along the two-edge a chain [x, y, blank tile] is shipped into nc_j. If at the last exit nc_j cannot be arranged as in B_2 then ignore all moves done for traversing the two-edge, swap the order of x and y in the chain and move the modified chain through the two-edge. Now nc_j can be arranged as in B_2 (see Theorem 1).

Finally, we need to show that nc_m can be arranged as in B_2. When nc_m is entered for the last time the whole board is arranged as in B_2 except for possible the area of nc_m. Call this board configuration B. Assume the board consisting only of the area of nc_m in B cannot be rearranged as the node nc_m in B_2. Then nc_m can be arranged as in B_2 except for one swap. But this is not possible since we reached from B_1 a configuration identical to B_2 except for one swap and by the construction of the boards B_1 and B_2 are reachable from each other.

The above completes the summary of the modifications to the procedures Shift that are needed. We next show that the modified procedures Shift require a total of at most $k = d(B_1,B_2)+u$ moves.

First observe that all moves done while traversing a zero-edge are necessary moves since each move shifts a tile from its location in B_1 toward its location in B_2 (see Figure 4). However the last move for each edge might not be necessary summing to at most $4m-3$ moves. Similarly, the seven-step sequences done when a chain was moved through a stripe of a two-edge consist of necessary moves: The tiles of the stripe and x and y always move towards their final location in B_2 (see Figure 5). The only exception to this are up to three seven-step sequences at the beginning and the end of each two-stripe. Since the number of two-stripes is $12m$ (see Figure 3) we follow that the total number of moves while traversing a zero-edge and while executing seven-step sequence is at most $d(B_1,B_2)+508m$.

All moves of the procedures Shift that we haven't accounted for yet occur while rearranging corners of two-edges and nodes (sometimes enlarged nodes). The number of corners of two-edges is $4m$ and per rearrangement of such a corner of size 3×3 we spent at most $p(3)$ moves (see Fact 1). There are $5m$ small nodes of size 3×3 which each are rearranged at most 5 times each, once for each in- and outgoing edge and one final rearrangement. The node size for each such rearrangement is at most 6×6. Thus all rearrangements of the corners and the small nodes cost at most $29m\,p(6)$.

The large nodes have size $4m$ and the total number of edges entering and leaving each of these nodes is $4m-2$. Furthermore, in FC up to $m-1$ swaps of the element tiles of a chain might be needed. The cost per rearrangement and swap is bounded by p(4m+3). Thus the total number of moves in the four Shift procedure is at most $d(B_1,B_2)+508m+29m\,p(6)+9m\,p(4m+3) = d(B_1,B_2)+u = k$.

This completes this subsection in which we have shown that if there is a truth setting solving the instance of 2/2/4–SAT then B_1 can be moved to B_2 in at most k moves. The next section completes the reduction.

4.3. Constructing a Solution to 2/2/4–SAT from a Solution to $nPUZ$

Let SOL be a sequence of at most k moves that transfers B_1 to B_2. We will show how to construct from SOL a truth assignment that satisfies the 2/2/4–SAT instance. Our method of proof is by showing that the blank tile and the element tiles are forced to move only along the edges

(except for negligible number of moves). Furthermore since k is not much larger than the number of necessary moves, this forces the number of tiles passing through each edge to be equal to the capacity of the edge, and it avoids a second move of tiles along an edge. Once we prove the above, we can eliminate the rules of the game. The instance of *nPUZ* corresponds to the instance of *REL* used in the reduction of the previous Theorem 2 (Claim 2). Also the solution *SOL* translates to a solution of this *REL* instance. The reduction of Theorem 2 (Claim 2) shows how to construct a truth assignment to the instance of *2/2/4–SAT* from the solution to the *REL* instance.

Since *SOL* consists of at most $k = d(B_1,B_2)+u$ moves, at most u unnecessary moves were done in *SOL*. This fact assures that during *SOL* the blank tile can only visit certain locations on the $n \times n$ board. Recall that the basic length unit l of the stripes is $u+16m+1$.

DEFINITION 1. The *vicinity of an edge* is obtained by widening all the stripes of the edges by l locations on each side. That is, the vicinities of edges of capacity c consist of stripes of width $2l+c+1$. The *vicinity of a small node* is the square of side length $2l+3$ obtained by enlarging the small node by l locations on each side. The *vicinity of TC (FC)* is built in two steps. Let *TC'* (*FC'*) be the smallest rectangle that contains *TC* (*FC* and all in- and outgoing stripes of *TC* (*FC*). *TC'* and *FC'* are the dotted rectangles of Figure 9. The *vicinity of TC (FC)* consists of the rectangle *TC'* (*FC'*) enlarged by l locations on each side. Observe that all vicinities are disjoint since the length of the stripes of the edges and the distance between any two nodes is at least $5l$ (see Figure 10).

LEMMA 1. *During SOL the blank tile is always in the vicinity of a node or an edge.*

PROOF. All locations outside the edges and the nodes contains the same tiles in B_1 and B_2. Any location L outside of the vicinities is distance at least $l+1$ away from any location of an edge or a node. Thus for the blank tile to get to location L in *SOL*, $l+1$ tiles were moved that are in the same location in B_1 and B_2, causing at least $l+1$ unnecessary moves. This is a contradiction since $l+1>u$. □

We next aim to describe what corresponds to moving along an edge in *SOL*. To do this we partition the vicinities of edges into three segments and show that the blank tile cannot move backwards through any of the segments.

DEFINITION 2. Let e be an edge going from its tail node n to its head node n'. We partition the locations of the vicinity of e into the three sets *entry*, *center*, and *exit*. The *entry* consists of all locations that are at distance at most l from the vicinity of n. The *exit* is defined similarly, except n is replaced by n'. The set of the remaining locations of the vicinity of the edge is called the *center* of the edge.

Observe that the entry and exit of an edge consist of a stripe of length l and the center up to two stripes, each of length at least l (see Figure 10).

LEMMA 2. *If in SOL the blank tile is located in the center of an edge, then the next node vicinity it will visit is the vicinity of its head node. If in SOL the blank tile is located in the vicinity of a node, then the next center of an edge it will visit belongs to an outgoing edge of the node.*

PROOF. It suffices to show that the blank tile cannot move backwards through the entry nor through the exit of an edge. We just show this for the entry of the edge, the proof for the exit is identical. Assume that the blank tile moved backwards through the entry. Let $Q = L_1, \cdots, L_q$ be the sequence of locations of the entry that were visited by the blank tile when traversing the entry from the center to the vicinity of the tail node. Let $L_{i_1}, \cdots, L_{i_l}$ be the

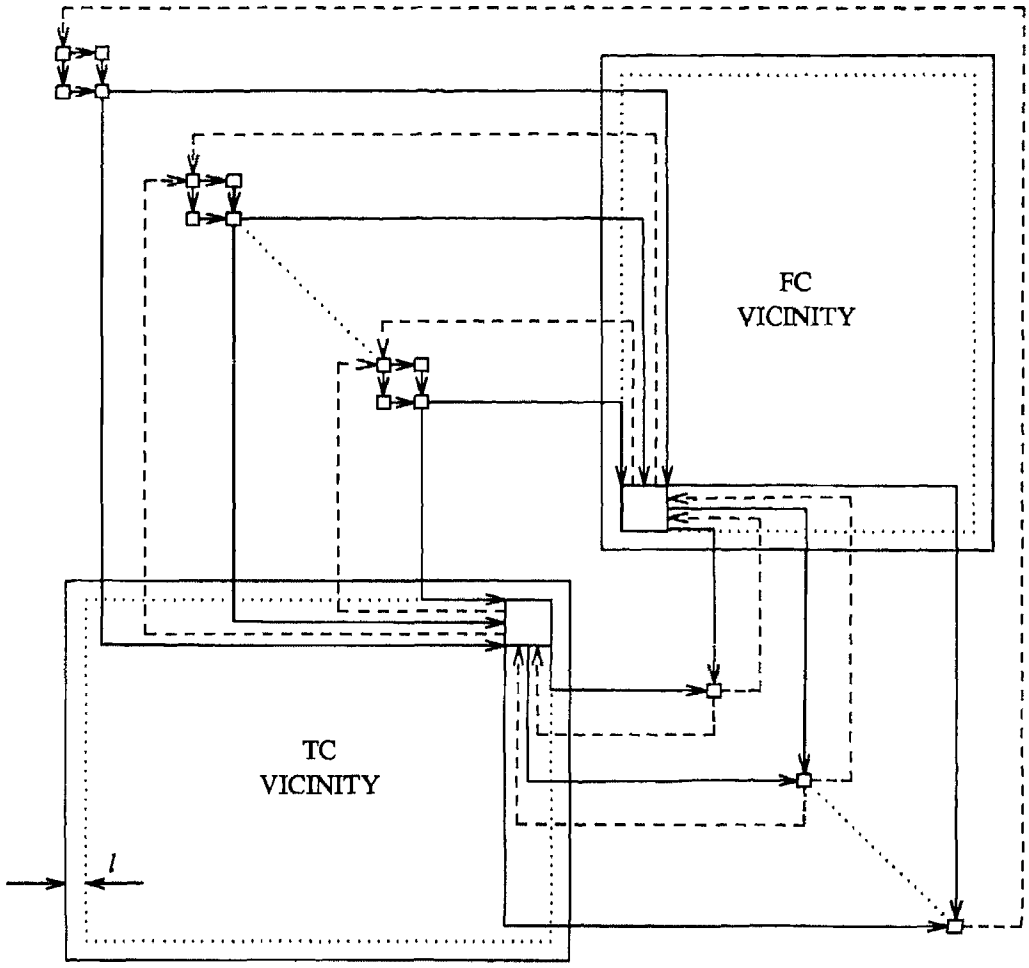

Figure 9: The vicinity of *TC* and *FC* nodes.

subsequence Q s.t. L_{i_j} is the first location of Q that is distance j away from the border line between the center and the entry ($1 \le j \le l$). Each location L_{i_j} of the subsequence is distinct and contains a different tile (denoted by t_{i_j}) before the blank tile traversed Q. By the definition of the subsequence it follows that all tiles $T = \{t_{i_j}\}$ traveled *forward* w.r.t. the direction of the edge when L_{i_j} was entered the first time in Q. If t_{i_j} was not an element tile, then the first move of the empty tile into L_{i_j} during Q must be an unnecessary move. Only element tiles can possibly travel in the direction of the entry by a necessary move. There are $8m$ such tiles and thus $l-8m$ unnecessary moves were performed while traversing the entry of an edge backward. This is a contradiction since $u > l-8m$. (Recall that $l=d+16m+1$. □

Traversing an edge in *SOL* corresponds to the blank tile traveling from the vicinity of the tail node to the vicinity of the head node through the center of the edge. We still need to show that each edge is traversed exactly once "carrying along" the right amount of element tiles.

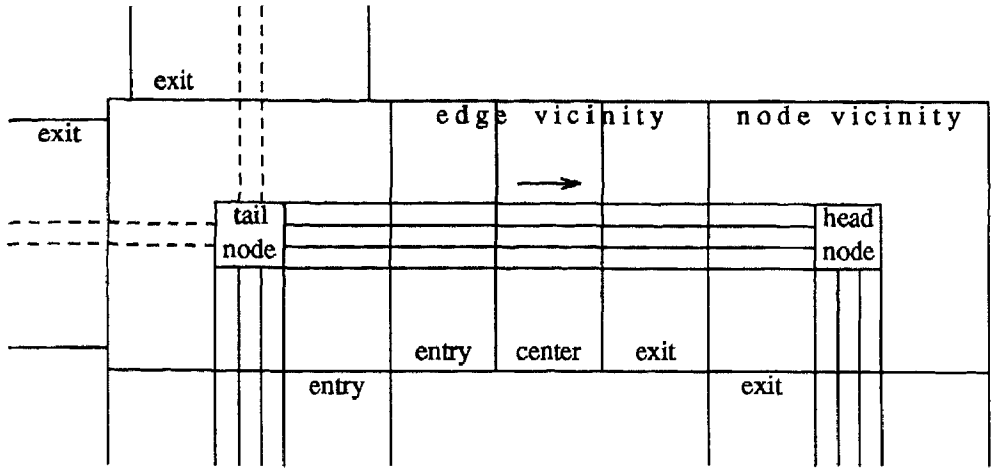

Figure 10: The vicinities around an edge $(top_i, \overline{nu}_i)$.

LEMMA 3. *Each edge is traversed exactly once.*

PROOF. By the previous lemma and since the stripes of the edges are rearranged in B_2 it follows that each edge must be traversed at least once. We first show that zero-edges are traversed exactly once. Consider the first sequence of moves of the blank tile through the entry of the edge. Let L_{i_j} be the first location during this sequence that is distance j away from the border line between the vicinity of the tail node and the entry area. Define the tiles t_{i_j} and T as in the previous lemma. When the locations L_{i_j} are visited the first time during the sequence then the l tiles of T are shifted backwards w.r.t. direction of the entry. If there is a second sequence that moves through the entry then a second set T' of l tiles are shifted backwards. The only tiles that can be shifted backwards in the entry area with necessary moves are the l tiles of the zero-edge located in the entry area of the board B_1 and possibly element tiles. The former tiles can only shift once backwards (see Figure 4) which amounts to l necessary moves. Each of the $8m$ element tiles may appear in both sets. This leaves $2l-l-16m > u$ tiles that were moved by unnecessary moves which is a contradiction.

We are left to show that the two-edges are traversed exactly once. There are exactly two edges going into the each diamond (except for the first diamond, but since we assume that the blank tile in B_1 is located in this diamond we can treat the first diamond as if there are two ingoing edges to it). These ingoing edges are zero-edges. Since the blank tile passes through these zero-edges exactly once (and through any edge at least once) it follows that the blank tile passes through all edges adjacent to the nodes of the diamond exactly once. For each nc_i clause node there are two ingoing edges and two outgoing edges. The outgoing edges are zero-edges; hence the blank tile passes through the ingoing edges exactly once. □

LEMMA 4. *When an edge of capacity c is traversed in SOL then number of element tiles that travel through the center while the blank tile moves through the edge is exactly c.*

PROOF. Let e be edge with capacity c. Execute *SOL* on B_1 but stop the sequence of moves right before the blank tile moves the first time from the entry area into the center area of e. Call the current configuration C_1. Continue executing *SOL* (the blank tile is traversing e) and stop *SOL* again right before the blank tile is moving into the vicinity of the tail node of e the

first time. Denote this second configuration by C_2 and the subsequence of *SOL* that moves C_1 to C_2 by S. The lemmas 2 and 3 imply that in *SOL* the blank tile cannot move back to the center from the configuration C_2. Thus the center in C_1 (C_2) must be arranged as in B_1 (B_2). From the definition of B_1 and B_2 it follows that during the subsequence S of *SOL*, a total of $c+1$ tiles are moved from the center into the entry and the same amount of tiles from the exit into the center (see figures 4 and 5). By Lemma 1 and by the definition of S, the blank tile never leaves the vicinity of the edge during *SOL*. Since the in- and outflow of tiles into the entry area must be the same, the blank tile together with exactly c tiles from the entry must travel through the center during S.

Assume one of the c tiles, denoted by x, was not an element tile. Since the center is composed of one or two stripes of length at least l, tile x traveled distance at least l while moving through the center. Let min be the distance of x from its location in B_1 to its location in B_2. If x appears on an edge in B_1 then min is at most 2 and if it appears in a node then min is at most $8m$ (the diameter of the large nodes). Thus the assumption that x is not an element tile leads to at least $l-8m>u$ unnecessary moves, which is a contradiction. □

This completes the proof of Theorem 3, since we showed that each edge is traversed exactly once with the right number of element tiles. Thus the instance of *nPUZ* (summarized in Figure 3) corresponds to the instance of *REL* (see graph of Figure 2). The sequence *SOL* translates to a sequence of edge traversals in the graph the solves the instance of *REL*. It was shown in Claim 2 of the previous reduction how to construct a truth setting for the instance of *2/2/4–SAT* from a solution to the instance of *REL*. □

COROLLARY 1. *Finding a solution in the* (n^2-1)*-puzzle that is an additive constant from the optimal solution is NP-hard.*

PROOF. Let r be the additive constant. We use the same reduction as above except l and u are both enlarged by r. This guarantees even for solutions *SOL* of length at most $k+r$ instead of at most k, lemmas 1, 2 and 3 are correct hence Lemma 4 holds. □

5. An Approximation Algorithm for the n^2-1-Puzzle

Since finding a shortest solution is NP-hard we would like to know how close a shortest solution for the (n^2-1)-puzzle can be approximated. As shown in the previous section finding a solution that is an additive constant larger than the optimum is also NP-hard.

In this section we sketch a polynomial algorithm that is guaranteed to produce a solution of length at most a constant times the length of the optimum (We first give a lower bound on the length of the optimum and then show that the approximation algorithm produces solutions of length at most a constant times the lower bound). Future research is needed to improve the lower bound and optimize the value of the multiplicative constant. The main purpose of this paper is to show that such a constant exists. As suggested by Ratner and Pohl [Ratner & Pohl, 1986], once a solution has been found whose length is guaranteed to be "close" to the optimum, then search methods should be used to locally optimize the solution.

5.1. A Lower Bound on The Number of Moves Required

Throughout this section we will assume that B_1 is the start configuration, B_2 the final configuration and *opt* the minimum number of moves for shifting B_1 to B_2. Also, whenever we write "approximation algorithm" we mean a polynomial approximation algorithm that

approximates the optimal solution by a multiplicative constant. For simplicity we assume that the blank tile resides in the same location in both configurations. The following lemma assures us that this assumption is valid.

LEMMA 5. *If there is an approximation algorithm where the blank tile is located in the same location in B_1 and B_2, then there is an approximation algorithm that does not assume that the blank tile is in the same location.*

PROOF. Let *opt* be as before and let d be the distance of the blank tile in B_1 and B_2. Clearly, $d \leq opt$. Let $AL_=$ be the approximation algorithm for the case when the blank tile is located in the same location and let c be its multiplicative constant. The following algorithm transforms B_1 to B_2: Move the blank tile from its location in B_1 to its location in B_2 and then use $AL_=$ to produce the final configuration B_2. Since the call to $AL_=$ requires at most $c\,(d+opt)$ moves, the total number of moves is at most $(2c+1)opt$. □

From now on we assume that the blank tile is located in l_{bt} in both configuration B_1 and B_2. We first proof a lower bound on *opt* which is the sum of two terms. The first term of the lower bound is the number of necessary moves $d(B_1,B_2)$, which can be expressed as follows. Express B_2 as a permutation of B_1 (defined in Section 2), decompose the permutation into disjoint cycles and let C be the set of all such cycles. For a cycle $C \in \mathrm{C}$, $d(C)$ denotes the sum of the distances of adjacent locations on the cycle (If C has size one then $d(C)=0$). Also $d(\mathrm{C})$ is the sum of $d(C)$ over all cycles $C \in \mathrm{C}$. Clearly $d(\mathrm{C}) = d(B_1,B_2) \leq opt$.

For the second term, let L be all locations at which B_1 and B_2 differ including the location of the blank tile l_{bt}. Let the cost of a spanning tree of a subset of L be the total sum of all distances of the edges. The value of *opt* is clearly lower bounded by the cost of a minimum cost spanning tree of any subset of L. We choose a subset that contains l_{bt} and exactly one location from each cycle. Combining the above leads to the following lower bound theorem.

THEOREM 4. *opt is at least as large as the half the sum of all distances of adjacent locations on the cycles of B_1 and B_2 plus the cost of a minimum cost spanning tree of any set of locations containing l_{bt} and exactly one location from each cycle.*

5.2. Sketch of an Approximation Algorithm

We first give an outline of the algorithm:

1. The algorithm finds a minimum cost spanning tree as described by the lower bound theorem.
2. It then "maps" the spanning tree and the cycles onto the board as follows. The *nodes* are the locations at which l_{bt} and the tiles of the cycles reside in the initial configuration. Any pair of nodes that are adjacent in the spanning tree or on a cycle is connected by *one* of the two possible orthogonal angles. This angle of locations is called the *edge* connecting the two nodes.
3. The blank tile traverses the nodes and edges of the spanning tree such that each edge is traversed exactly once in each direction. When an edge is traversed the first time the blank tile moves each tile encountered one step "backwards" (see Figure 4). The second traversal of the edge (in opposite direction) undoes the changes of the first. By the definition of the tree it follows that the blank tile encounters at least one tile from each cycle during the traversal. Whenever during the traversal of the tree the blank is adjacent to a tile of new cycle, then the approximation algorithm "fixes" the cycle of that tile by rotating the tiles of the cycle along the edges of the cycle to their location in the final configuration (this is described in more detail below). After the cycle is fixed the traversal of the tree is continued.

Fixing a cycle: Let $C = (t_0, t_1, \ldots, t_{c-1})$ be the cycle to be fixed and l_i be the location at which t_i resides in B_1. Assume the blank tile is adjacent to location l_0. The blank tile first moves with t_0 along (l_0, l_1), then with t_1 along (l_1, l_2), and so forth until t_{c-2} is moved from l_{c-2} to l_{c-1}. Secondly, the blank tile moves with the final tile t_{c-1} from l_{c-1} to l_0 via the path $l_{c-2}, l_{c-3}, \cdots, l_0$.

Call the locations of the nodes and edges of the spanning tree and the cycles the *primary locations* of the board. We now sketch how the blank tile (denoted by ∇) moves along a directed edge with some tile x. Assume on an edge the following tiles are in four adjacent locations: $\nabla, x, \alpha_1, \alpha_2, \alpha_3$, where the location on which ∇ resides is closer to the tail of the edge. There are simple macros that arrange the tiles on the four locations to $\alpha_1, \nabla, x, \alpha_3, \alpha_2$ and then to $\alpha_1, \alpha_2, \nabla, x, \alpha_3$. During the execution of such a macro the blank tile moves outside of the four locations but the macros have the property that all tiles other than the ones in the four locations are at the same location before and after the execution of a macro. For lack of space we omit an exact description of the macros but their existence of is implied by Theorem 1. Each of them only requires a constant number of moves and advances x by exactly one location w.r.t. the direction of the edge. Applying the macros iteratively moves the blank tile and x along the edge. Observe that the tiles of the edges (the α_i's) are shifted by exactly two locations backwards while moving the blank tile with x through the edge. All other tiles remain in the same location. This change is undone by moving backwardly through the edge with some other tile. In the above fix procedure each edge of the cycle is either traversed once in each direction or not at all.

In general, the entire approximation algorithms consist of a sequence of macros and simple shifts of the blank tile on primary locations. A small number of additional macros are needed to deal with the movement through the nodes. The blank tile only temporarily enters non-primary locations during the execution of the macros. Between macros all non-primary locations contain the same tiles as in the initial and final configuration.

This completes a rough sketch of the approximation algorithm. There are two subtle problems that we still need to address. First, because of the reachability constraints of (n^2-1)–puzzle , cycles of even size can only be fixed up to a swap of two tiles (Theorem 1). Second, a node l_i of the cycle might appear on an edge of the same cycle (see Figure 11). While this edge is traversed, t_j is moved by two locations away from l_j. This may happen many times and t_j might be pushed "far away" from l_j. Call the nodes of the cycle that appear on the edges of the cycle *early* nodes.

To fix the first problem observe that by Theorem 1 the number of even cycles is even. When the first even cycle is fixed up to a swap, we can carry the swap along the spanning tree (possibly fixing odd cycles on the way) until we reach the next even cycle. Now this cycle can be fixed completely since we entered it with a swap. This process is iterated. The mention parity property guarantees that at the end of the whole traversal no swaps are left. Carrying a swap along is done iteratively using a macro of a constant number of moves that arranges three adjacent location on an edge from $\nabla, \alpha_2, \alpha_1, \alpha_3$ (where α_1 and α_2 are swapped) to $\alpha_1, \nabla, \alpha_3, \alpha_2$ (where α_2 and α_3 are swapped). Again the existence of such a macro is implied by Theorem 1.

To fix the second problem we modify the traversal of the cycle. For the description of the modification it is easier to denote a cycle by the sequence of location (i.e. the nodes) on which the tiles of the cycle reside. We first explain how the fixing process is done using the cycle $(l_0, \cdots, l_5)$ of Figure 11. Note that in this cycle the node l_3 is early, i.e. it appears on the edge (l_0, l_1). To fix the cycle we begin moving t_0 along the edge (l_0, l_1). However we stop this

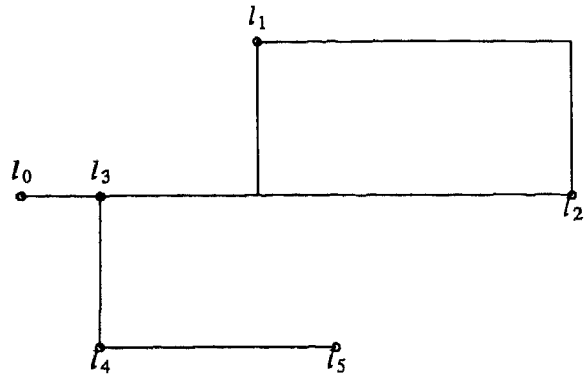

Figure 11: An (early) node of a cycle appears on an edge of the cycle.

process right when t_0 is at location l_3. At this point the initial tiles of the edge (l_0,l_1) including the tile t_3 were shifted two locations backwards on the edge and the second location before l_3 contains t_3 followed by the blank tile and then t_0 in l_3. We are now ready to fix the cycle (l_3,l_1,l_2) using the previous method (Note that t_0 is in l_3). At the beginning of this fixing process of the smaller cycle the blank tile is adjacent to t_0 in l_3 and at the end the blank tile is again in the same location and the tiles t_0, t_1 and t_2 are in their final location. Then move with t_3 along the edge (l_3,l_4) fixing the cycle (l_3,l_4,l_5). Finally shift t_5 from l_3 to l_0. The total path of the blank tile used for fixing the cycle of Figure 11 is given in Figure 12.

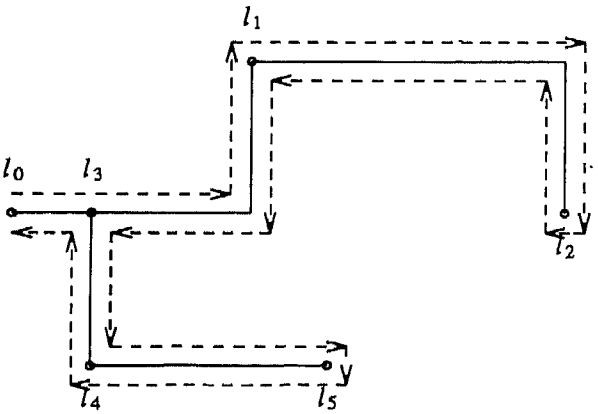

Figure 12: The path used to fix the cycle of Figure 11.

In general, the subcycles might again contain early nodes on their edges. Thus the fixing process might again split cycles. We will fix the subcycle of the cycle in a depth-first-search fashion. Formally, let $C = (l_0,l_1,\ldots,l_{c-1})$ be the cycle that we are to fix and while traversing the path $l_0,\cdots,l_{c-1}$ let l_j be the first early node that appears on some edge (l_i,l_{i+1}) of the cycle. Note that $j>i$. We begin fixing the whole cycle by moving the blank tile around the cycle, at each point carrying along the proper tile. This process is stopped when t_i is shifted onto l_j while attempting to move with t_i along (l_i,l_{i+1}). As above, the location before l_j now contains the blank tile and one before that t_j. As a next step we recursively fix $(l_j,l_{i+1},\cdots,l_{j-1})$. Recall that before this call t_i resides in l_j and after the call the tiles $t_i,\cdots,t_{j-1}$ are at their final

location. We continue moving with t_j along (l_j,l_{j+1}) fixing the cycle $(l_j,l_{j+1},l_{j+2},\cdots,l_{c-1})$. Finally, we shift l_{c-1} from l_j to l_0. Note that whenever a subcycle is fixed then this might cause nested calls for fixing subcycles. This completes the sketch of the algorithm. A more detailed description is given in [Ratner, 1986] and omitted here.

It is easy to see by an induction on the size of the cycle that the recursive fixing procedure is correct. The correctness of the whole algorithm follows now from the fact that during the traversal of the tree the tiles on the path from l_{bt} to the current location are shifted one step backwards. All cycles encountered so far are corrected in the sense that the tiles of these cycles (that are not on the path) are rotated to their place in B_2. Otherwise the current board is identical to B_1.

The total number of moves used by the approximation algorithm is no more than a constant times the lower bound, since per node and per location of an edge of the tree and the cycles only a constant number of moves is required. The time to find a minimum cost spanning tree is $O(|C|^2)$ [Prim, 1957]. Except for finding the spanning tree, the run time of the approximation algorithm is linear in the lower bound. Thus we conclude with the following theorem.

THEOREM 5. *There is a polynomial approximation algorithm for* (n^2-1)–puzzle *that produces a solution of length at most constant times the length of the optimum solution.*

5.3. Optimizations of the Approximation Algorithm

We complete the paper by discussing various optimizations of the sketched approximation algorithm:

1. Observe each cycle contributed a location to the node set of the minimum cost spanning tree. By modifying the standard minimum cost spanning tree algorithms [Prim, 1957] one can choose the representatives of the cycles such that the cost of the produced tree is minimized [Ratner, 1986].
2. To fix the second problem (nodes occurred on edges) we broke the cycle into two cycles and recursively fixed each of the smaller cycles. A similar case is given in figures 13 and 14, where two edges of the cycle intersect on the board. We move t_0 to the intersection point l, then recursively fix the cycle (l,l_1,l_2) before continuing to fix the cycle (l_0,l_3,l_4). It turns out that splitting cycles using intersections between edges saves moves.
3. Similar saving can be obtained when two different cycles intersect.
4. In the above we observed that intersections of cycles result in shorter move sequences. The algorithm can embed the edges of the cycles and the spanning tree in the board s.t. a large number of intersections occur.
5. There are other basic ways of fixing a non-intersecting cycle: For example move twice backwards around the cycle with only the blank tile, and then once forwardly, at each point carrying along one tile.

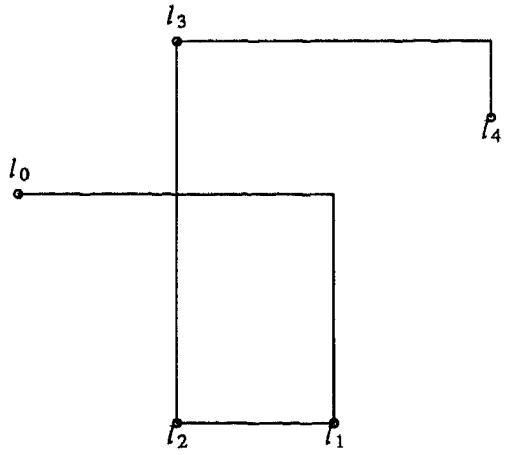

Figure 13: A cycle in which two edges intersect.

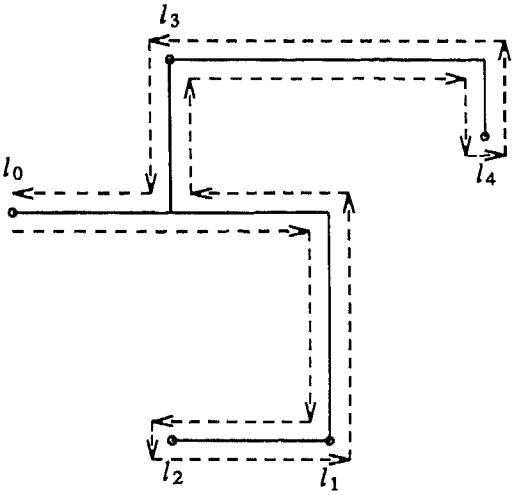

Figure 14: A shorter path that fixes the cycle of Figure 13.

Acknowledgements

We would like to thank Prof. Ira Pohl for suggesting to us that *nPUZ* might be NP-complete.

The second author gratefully acknowledges the support of ONR grants N00014-86-K-0454 and NOOO14-85-K-0445.

References

Doran, J., Michie, D. (1966). Experiments with the graph traverser program. *Proceedings of the Royal Society (A)*, **294** pp. 235-259.

Fiat, A., Moses, S., Shamir, A., Shimshoni, I., Tardos, G. (1989). Planning and learning in permutation groups. *Proceedings 30th IEEE Symposium on Foundation of Computer Science*, to appear.

Fraenkel, A.S., Garey, M.R., Johnson, D.S., Schaefer, T., Yesha, Y. (1978). The complexity of checkers on an N by N board. *Proceedings 19th IEEE Symposium on Foundation of Computer Science*, pp. 55-64.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. Ph.D. Thesis, Carnegie-Melon University.

Goldberg, A., Pohl, I. (1984). Is Complexity Theory of Use to AI? *Artificial and Human Intelligence*, Elsevier Science Publishers.

Goldreich, O. (1984). Finding the shortest move-sequence in the graph-generalized 15-puzzle is NP-hard. Labaratory for Computer Science, MIT, unpublished manuscript.

Hopcroft, J.E., Schwartz, J.T., Sharir, M. (1984). On the complexity of motion planning for multiple independent objects; PSPACE_Hardness of the "Warehouseman's Problem". *The International Journal of Robotics Research* 3 pp. 76-88.

Korf, R.E. (1985). Learning to Solve Problems by Searching for Macro-Operators. *Research Notes in Artificial Intelligence* **5**, Pitman Advanced Publishing Program.

Korf, R.E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* **27** pp. 97-109.

Kornhauser, D., Miller, G., Spirakis, P. (1984). Coordinating pebble motion on graphs, the diameter of permutation groups, and applications, *Proceedings 25th IEEE Symposium on Foundation of Computer Science*, pp. 241-250.

Lawler, E.L., Lenstra, J.K., RinnooyKan, A.H.G., Shmoys, D.B. (1985). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* John Wiley & Sons.

Lichtenstein, D., Sipser, M. (1978). Go is Pspace hard. *Proceedings 19th IEEE Symposium on Foundation of Computer Science*, pp. 48-54.

Loyd, S. (1959). *Mathematical Puzzles of Sam Loyd.* Dover, New York.

McCoy, N.H., Berger, T.R. (1972). *Algebra: Groups, Rings, and Other Topics* Boston: Allyn and Bacon.

Pearl, J. (1984). *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley.

Pohl, I. (1971). Bi-directional search. B. Meltzer, D. Michie, eds. *Machine Intelligence* **6** New York: American Elsevier, pp. 127-140.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. *Proceedings Third International Joint Conference on Artificial Intelligence.* Stanford, CA.

Pohl, I. (1977). Practical and theoretical considerations in heuristic search algorithms. B. Meltzer, D. Michie, eds. *Machine Intelligence* **8** New York: American Elsevier pp. 55-72.

Politowski, G. (1986). On construction of heuristic functions. Ph.D. Thesis, University of California at Santa Cruz.

Prim, R.C. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal* **36**, pp.1389-1401.

Ratner, D. (1986). Issues in theoretical and practical complexity for heuristic search algorithms. Ph.D. Thesis, University of California at Santa Cruz.

Ratner, D., Pohl, I. (1986). Joint and *LPA**: combination of approximation and search. *Proceedings Fifth National Conference on Artificial Intelligence*, Philadelphia, PA.

Rendell, L.A. (1983). A new basis for state-space learning systems and a successful implementation. *Artificial Intelligence* **20** pp. 369-392.

Schaefer, T.J. (1978). The complexity of satisfiability problems. *Proceedings 10th Annual ACM Symposium on Theory of Computing*. New York.

Appendix

THEOREM 6. *2/2/4–SAT is NP-complete.*

PROOF. We will reduce *2/2/4–SAT* to *1/3–SAT*, a known NP-complete version of the satisfiability problem [Schaefer, 1978]:

INSTANCE. A set U of variables and a collection C of clauses over U such that each clause $c \in C$ has exactly three literals ($|c| = 3$).

QUESTION. Is there a truth assignment for U such that each clause in C has exactly one true literal ?

Let $U = \{u_1, u_2, \ldots, u_n\}$ be a set of variables and $C = \{c_1, c_2, \ldots, c_p\}$ be a set of clauses which are an arbitrary instance of *1/3–SAT*. In the following four steps we construct the corresponding instance of *2/2/4–SAT*: a collection C^* of $7p$ four literal clauses over a set V of $7p$ variables. Each $v \in V$ appears exactly twice negated and twice unnegated in the clauses of C^*.

STEP 1. Rename each appearance of a literal in each clause of C with a unique name (variables $v_{i,j}$, for $1 \le i \le p$ and $1 \le j \le 3$) and add a new variable w_i to each clause. Thus as a result of this step we have a set V_1 of variables and a set C_1 of clauses such that:

$V_1 = \{ v_{i,j} \mid 1 \le i \le p, 1 \le j \le 3 \} \cup \{ w_i \mid 1 \le i \le p \}$

$C_1 = \{ (v_{i,1}, v_{i,2}, v_{i,3}, w_i) \mid 1 \le i \le p \}$

STEP 2. The clauses of this step assure that all $v_{i,j}$ that correspond to the same literal u_k in C have the same assignment. Also all $v_{i',j'}$ that correspond to the literal $\bar{u}_k$ must have the opposite assignment than the $v_{i,j}$ variables that correspond to the literal u_k.

Let l be a literal in the *1/3–SAT* instance.

$Occur(l) := \{v_{i,j} \mid l$ is the j–th literal in $c_i\}$ and

$a_i := |Occur(u_i)|, \quad b_i := |Occur(\bar{u}_i)|, \quad s_i := \sum_{j=1}^{i-1}(a_j + b_j).$

Note that a_i (b_i) is the number of occurrences of u_i (respectively $\bar{u}_i$) in the formula of the *1/3–SAT* instance. Let $Occur(u_i) := \{x_1, x_2, \cdots, x_{a_i}\}$ and $Occur(\bar{u}_i) := \{y_1, y_2, \cdots, y_{b_i}\}$. Now we construct, for each $u_i \in U$, a chain of $(a_i + b_i)$ 4 literals clauses, called $C_{2,i}$. This chain consists of four subsets $C^{(k)}{}_{2,i}$, for $1 \le k \le 4$. In each new clause we introduce a new variable. The chain is:

$C^{(1)}{}_{2,i} = \{ (x_j, \bar{x}_{j+1}, z_{s_i+j}, \bar{z}_{s_i+j}) \mid 1 \le j \le a_i - 1 \}$

$C^{(2)}{}_{2,i} = \{(x_{a_i}, y_1, z_{s_i+a_i}, \bar{z}_{s_i+a_i})\}$

$C^{(3)}{}_{2,i} = \{ (\bar{y}_j, y_{j+1}, z_{s_i+a_i+j}, \bar{z}_{s_i+a_i+j}) \mid 1 \le j \le b_i - 1 \}$

$C^{(4)}{}_{2,i} = \{(\bar{y}_{b_i}, \bar{x}_1, z_{s_i+a_i+b_i}, \bar{z}_{s_i+a_i+b_i})\}$

If $a_i = 0$ then $C^{(1)}{}_{2,i} = C^{(2)}{}_{2,i} = \phi$ and $C^{(4)}{}_{2,i} = \{(\bar{y}_{b_i}, y_1, z_{s_i+b_i}, \bar{z}_{s_i+b_i})\}$.

If $b_i = 0$ then $C^{(3)}{}_{2,i} = C^{(4)}{}_{2,i} = \phi$ and $C^{(2)}{}_{2,i} = \{(x_{a_i}, \bar{x}_1, z_{s_i+a_i}, \bar{z}_{s_i+a_i})\}$.

In summary this step adds the following variables and clauses:

$V_2 = \{ z_j \mid 1 \le j \le 3p \}$

$C_2 = \bigcup_{i=1}^{n} C_{2,i}$ where $C_{2,i} = \bigcup_{k=1}^{4} C^{(k)}{}_{2,i}$.

STEP 3. This step forces the variables w_i for $1 \le i \le p$, to posses the same truth assignment in the 2/2/4–*SAT* instance. We use the same technique as in Step 2 and add the following p clauses:

$C_3 = \{ (w_i , \bar{w}_{i+1} , z_i , \bar{z}_i) \mid 1 \le i \le p-1 \} \cup \{ (w_p , \bar{w}_1 , z_p , \bar{z}_p) \}$

STEP 4. We now add clauses in order that each literal will appear exactly twice. Note that each of the literals $v_{i,j}$, w_i , z_i , and $\bar{z}_i$, for $1 \le i \le p$ and $1 \le j \le 3$, have already appeared twice. The literals $\bar{v}_{i,j}$, for $1 \le i \le p$ and $1 \le j \le 3$, and the literals $\bar{w}_i$, for $1 \le i \le p$, have appeared only once in the previous steps. Therefore we add another set of clauses, called $C_{4,1}$, which is as exactly as C_1 except that all the variables in the clauses are negated: $C_{4,1} = \{ (\bar{v}_{i,1} , \bar{v}_{i,2} , \bar{v}_{i,3} , \bar{w}_i) \mid 1 \le i \le p \}$

The literals z_i and $\bar{z}_i$, for $p+1 \le i \le 3p$, have appeared only once (in C_2). They appear a second time in the following set of clauses, called $C_{4,2}$:

$C_{4,2} = \{ (z_{p+i} , \bar{z}_{p+i} , z_{2p+i} , \bar{z}_{2p+i}) \mid 1 \le i \le p \}$

$C_4 = C_{4,1} \cup C_{4,2}$

The four steps give the corresponding instance of 2/2/4–*SAT*, where

$C^* = C_1 \cup C_2 \cup C_3 \cup C_4$ and $V = V_1 \cup V_2$.

Note that each $v \in V$ appears exactly four times, twice negated and twice unnegated. We only sketch the proof that the instance of 1/3–*SAT* has a valid truth setting (one true in each clause) iff the instance of 2/2/4–*SAT* has a valid truth setting (two true per clause). An extended proof and examples are given in [Ratner, 1986].

Given a truth setting which is a solution to the instance of 1/3–*SAT*. Assign all literals in *Occur* (l) the same value as l and set all variables w_i and z_j to true. It is easy to check that the constructed truth assignment for the instance of 2/2/4–*SAT* contains exactly two literals per clause.

Finally given a truth setting for the instance of 2/2/4–*SAT*. Note that the complementary truth setting also contains exactly two true literals per clause. Pick the truth setting that sets w_1 to true. The clauses of C_3 assure that all the w_i are set to true as well. Thus in the clauses of C_1 exactly one of the literals $v_{i,1}$, $v_{i,2}$, $v_{i,3}$ is set to true. The clauses of C_2 force all variables in *Occur* (l) to have the same value and all variables in *Occur* ($\bar{l}$) have the same but the opposite value. We conclude that by assigning the literal l the same value as the members of *Occur* (l) we get a solution to the instance of 1/3–*SAT*. □