Comparing AI Search Algorithms and Their Efficiency When Applied to Path Finding Problems

By

Erdal Kose

A dissertation submitted on the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York
2012

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy

Dr. Danny Kopec

_____ _____
Date Chair of Examining Committee

Dr. Ted Brown

_____ _____
Date Executive Officer

Dr. Paula Witlock

_____

Dr. Subash Shankar

_____

Dr. Stathis Zachos

_____

Supervision Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

Comparing AI Search Algorithms and Their Efficiency When Applied to Path Finding Problems

By

Erdal Kose

Advisor: Professor Dr. Danny Kopec.

Various Artificial Intelligence (AI) search algorithms have been investigated and classified as unidirectional or bidirectional. We start our discussion by presenting certain unidirectional and bidirectional search algorithms (BDS). We continued our study by presenting contributions to the field of search algorithms in AI. The focus of this research is the study of the bidirectional search and certain classes of AI problems and some new approaches to the domain of AI search algorithms have been explored. The second contribution of this research is to compare problem representations and to exploit built-in features of diverse programming paradigms. The programing paradigms have been classified by the problem domains which they might be more suitable for. This has been justified by the fact that the evaluation of search algorithms is a well-studied area of Artificial Intelligence (AI). However evaluation of the performance of programming paradigms when applied to search algorithms has not been studied well. We conclude our discussion with experimental results and more detailed information about our implementations.

Contents

List of Figures

List of Tables

Abbreviations

AI: Artificial intelligence.

IDA*: iterative Deeping Search

A*: Best First Search

IA* : Indexed A* Search

DFS: Depth First Search

BFS: Breadth First Search

IBFS: Indexed Breadth First Search

BDS: Bidirectional Search

PS* Perimeter Search employed A* algorithm to search from start toward goal state

IPS* Perimeter Search employed IDA* algorithm to search from start toward goal state

BIDA*: An Improvement Perimeter Bi-directional Search algorithms.

BAI: Bidirectional A* -IDA*

RTBS: Real Time Bi-directional Search.

RTUS: Real Time Unidirectional Search.

SFBS: Frontier Bidirectional Search.

1. Background and Research Goals

Search is a problem solving mechanism in AI, and the choice of search procedure is a prescription for determining in what order the nodes in a problem are to be generated and examined. In blind search techniques, this procedure is achieved by searching for a goal without using any information. With heuristic search techniques, partial information about the problem domain is used to guide the search from a start node towards a goal node. Unidirectional searches initiate a search from a start node towards a goal node without changing the search direction. The algorithm terminates if a solution is found or no solution has been found. If the algorithm employed heuristics it is called a heuristic search. If it does not use any heuristics then it is called a blind search algorithm. The bidirectional search combines two searches; one from start node towards a goal node and the second search from a goal node toward a start node, or vice versa. Whenever two searches collide a solution is found. The advantage of the bidirectional search is that it reduces the search domain's complexity in terms of number of nodes needed to be explored from b^d to $b^{d/2}$, where *b* is the branching factor, and *d* is the solution depth. In theory this is a big advantage, but experimental results show that most of the time the two searches fail to meet at the halfway point of search. Usually they will pass each other and instead of one search, the BS ends up being two unidirectional searches. The details of problems with the BDS are investigated more deeply in the following sections.

Unidirectional search algorithms have been studied comprehensively for decades; however the BDS has been relatively neglected since its invention by Dantzig in 1960

[7]. Subsequently, Nicholson in 1966 [27], and then Pohl in 1971 [30] independently explored the BDS more deeply.

The most regular form of the Breadth First Search (BFS) employs two lists for maintaining a solution, called the Open and Closed List. The Open List is for storing the frontiers of a solution tree, and the Closed List is for storing the nodes which have already been explored. There are two reasons to maintain the Closed List: First, to prevent duplication. While the search proceeds, cycles may occur. Most search algorithms employ a Closed List to prevent cycles and duplicate nodes. Secondly, the Closed List is used to build solution paths after a goal node has been reached. Here we will present new research introducing a new type of BFS which eliminates the Closed List. Instead of using a Closed List, our new approach uses indices to generate a solution path after a goal node has been reached. It also uses some heuristics (see section 10) to avoid duplicate nodes and cycles. Then we adopt the same indexing idea, eliminating the Closed List, to the A* search (see section 5.3). There are two main advantages of eliminating the Closed List:

1. Space is saved.
2. The process of generating a solution path from the index to the goal node is simplified.

We also investigated the domain of AI search problems. We believe that if a certain class of problems is implemented with the most suitable programing paradigm, it may reduce the programing overhead and may also reduce the time and space used by the

implementation. Problem representation methods within diverse programing paradigms are also investigated.

1.1. Dissertation Overview

The rest of the document is organized as follows:

Chapter 3: Search Algorithms in Artificial intelligence (AI)

This chapter provides an overview of the problem representations and an overview of AI search algorithms.

Chapter 4: Blind Search Algorithms

In this chapter we describe the main form of blind search algorithms such as Depth-First-Search (DFS), Breadth-First-Search (BFS), and Depth-First-iterative Deepening (DFID)

Chapter 5: Heuristic (Informed) Search Algorithms

We introduce heuristic search algorithms in this chapter. The Hill Climbing Search, Best-First-Search, A*, Iterative Deepening A* (IDA*), Recursive Best First Search (RBFS), Beam Search and Branch and Bound algorithms are described briefly.

Chapter 6: Bidirectional Search Algorithms:

In this chapter various Bidirectional Search Algorithms (BDS) are presented. We investigated the bidirectional search algorithms starting from Pohl's "Front-to- End Evaluation" approach, which is considered the first BDS algorithm, to the "Single-

Frontier Bidirectional Search" approach which has been published by Felner, Mondenhauer, Sturtevant, and Schaeffer in 2010 [10].

Chapter 7: Probabilistic Methods:

We summarize probabilistic methods in this chapter.

Chapter 8: AI Problem Domains:

This chapter discusses in detail AI problem domains and the approaches to solve them. The Klotski Puzzle and Fifteen Puzzles are introduced.

Chapter 9: State Space Search and Cost Functions:

In this chapter, we introduce the state space and cost functions we have employed to solve puzzles.

Chapter 10: Newer Types of Search Algorithm:

We present two newer types of AI search algorithms. We call them the Indexed Breadth First Search Algorithm (IBFS) and Indexed A* (IA*).

Chapter 11: Programming Language Paradigms and Their impact on the Efficiency of AI Search Algorithms

This chapter summarizes the three programming paradigms and describes their approaches to solving AI problems.

Chapter 12: Comparison of AI Programming Paradigms:

In this chapter we compare diverse programing paradigms when Applied to Single-Agent-Path-Finding Problems.

Chapter 13: Implementation and Experimental Results

Discusses experimental results. First we compare the implementations with C++ with Java. Then we compare our newer approaches' results with BFS and A* algorithms

Chapter 14: Conclusions and Future Work

This chapter discusses the contributions of this research and directions for future work.

2. Search Algorithms in AI

Each node in a search tree is a state of a search problem; a *state space* (see section 9) is a set of states and operators that maps states to states. Search problems can be represented with a graph $G(N,V)$, where *N* is a set of nodes such that $N = \{1, 2, 3, 4, \dots n\}$, and *V* is a set of vertices connecting nodes such that $V = \{a_1, a_2, a_3 \dots a_m\}$. A vertex is $a_t = (i,j) \in V$. A path *P* is a subset of *N* where each node of the path is connected, e.g. $P = \{n_1, n_2, n_3 \dots n_n\}$. The cost of travel from vertex *i* to *j* is $c(i,j)$. The purpose of search algorithms is to find a desirable path that maps a start node to a goal node by applying suitable operations to each state of a search tree. A list of AI search algorithms is presented in the figure 2.1.

In the following sections some well-known blind search algorithms are summarized, such as the DFS, BFS, and DFSID. After that we describe some well-known heuristic search algorithms, such as Hill Climbing, A*, IDA. Then we look at backward search techniques. Following the unidirectional search algorithms, we briefly described the class of BS algorithms. We summarize Pohl's BDS algorithm [34] "*Front to End Evaluation*", and then we describe its successors such as "The *Front to Front Approach (BHFFA)*" by de Champeaux & Sint 1977 [3] , BHFFA2 by de Champeaux 1983 [4], PS* by Dillenburg & Nelson in 1994 [8], BIDA* by Manzini 1995 [27, 28], GBS and DBS by Kaindl Kainz 1997 [13], Divide- and Conquer Bidirectional Search by Korf 1999 [19] and Single Frontier Bidirectional Search (SFBS) by Felner, Moldenhauer, Sturtevan, Schaeffer in 2012 [10].

Other search techniques will not be covered here since they can be found in any AI text book or from our references.

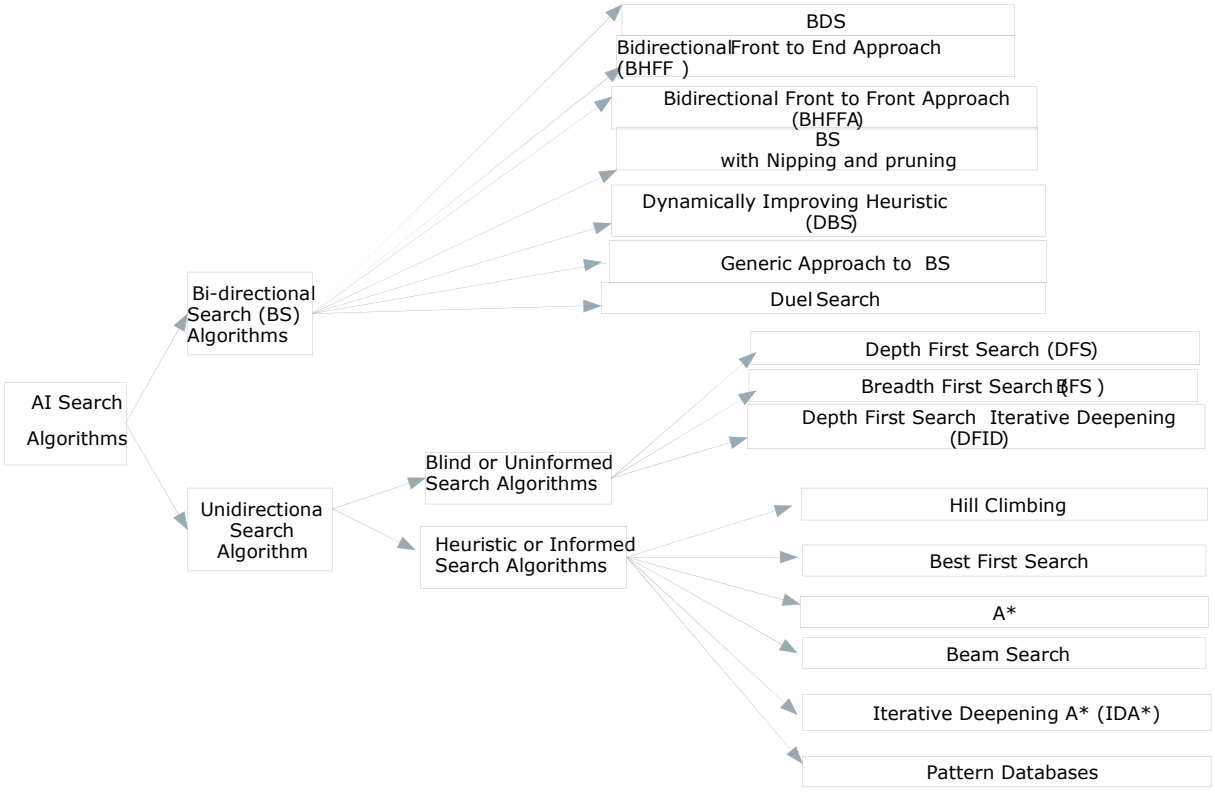


Figure 2.1. Search Algorithms for AI.

3. Blind Search

Blind search techniques are also called uninformed search algorithms, because they have no information about the search domain in which they are being applied. The only information available to them is the identification of a node as either a goal node or not. They follow a prescribed procedure until a goal state has been reached or no solution has been found.

Unidirectional blind search methods look for a goal node from one frontier[1] towards a goal node without using any information. For example, if we are searching through a state space graph of a problem for a target node, *t,* from a start node, *s, w*e could start searching from *s* until we reach that goal node *t*, if it is exists. At the end of the search if a solution is found, the algorithm will return a solution path, otherwise it will return false. This search technique is called *forward searching*. If a search is started from a target node to a goal node then the search technique is called *backward search*.

Blind search could also be established from both directions; one search from a start node towards a goal, and the other search from a goal node to a start node. Both searches are running in parallel. These classes of search methods are called Blind BDS Algorithms.

[1] The nodes of a search tree that are examined for a solution path (At the first level of search tree the frontier node is the root node).

3.1. Depth-First Search

Depth-First-Search (DFS) [25, 33] attempts to traverse as deeply into a tree as quickly as possible. Whenever a choice is presented, the leftmost or the right most branches are selected, and after it reaches the leftmost or the rightmost leaf of the tree, it will examine the leaf and backtrack to the parent of the leaf and explore the other children. It keeps examining nodes and leaves, backtracking until a goal is reached or no goal has been found.

DFS maintains two lists; Open and Closed Lists. All the examined nodes are kept in the Closed List. The explored nodes which have not yet been examined are kept in the Open List. These nodes may be examined or explored later on. The Open List is a first in last out (FILO) list. Each element of the Open List is examined recursively, whenever a goal has been found or the Open List is empty, the algorithm stops.

Lemma: DFS is complete if the depth of the tree is limited, otherwise it will get lost in the depth of the tree.

Proof: let L be a FILO list, and the depth of the tree is ***d***. If the leftmost node is ***n*** and when ***n*** has been examined, all the ancestors of ***n*** must be in L, and ***n*** is on the top of list. If ***n*** is not a goal then n will be popped off the list and all siblings of ***n*** (if any) will be pushed onto L to be examined; otherwise ***n'***s parent will be examined. The search backtracks until list L is empty. This procedure will continue until a goal has been found or the list L is empty.

The advantage of DFS is that its space requirement is only linear with respect to search depth. However, it may also get lost in a deeper and deeper search. In an infinite tree or a problem which has a large solution domain (such as chess) it will get lost in the left-most path.

3.2. Breadth-First Search (BFS)

A second blind search approach is the Breadth-First Search [24, 29]. It differs from DFS only in how it explores the nodes of the search tree. The BFS explores nodes in the order of their distance from the root, generating nodes level by level from top to bottom until a solution has been found. So that all the nodes at level *j* are expanded before the nodes at level *j+1* are expanded.

BFS is guaranteed to find a solution if any solution is available. It is also guaranteed to find the shortest path to the root. Akin to DFS, BFS maintains two lists Open and Closed, and the functionality of the lists is the same as the DFS lists, except the *Open List* is a last in first out (LIFO) list.

The disadvantage of BFS is that it may suffer from memory problems, because it examines all the nodes near the root. Hence the number of nodes generated could quickly become unmanageable. The number of nodes generated is a function of the branching factor and depth function. If the branching factor is b and the depth factor is d, then the number of nodes at depth d will be b^d.

For problems which have a deep goal, the DFS algorithm will generate a solution path more efficiently than the BFS, but for problems which have a solution near the root, the BFS is better than the DFS.

3.3. Depth First Iterative Deepening (DFID)

The DFS and BFS algorithms are not suitable for most search problems, as we have described above. DFID [20] combines the best features of DFS and BFS in one. DFID [20] first performs a DFS to depth one. If a solution has not been found, it will start another DFS to depth two. It will repeat this procedure, increasing depth *d* by one until a solution is found. It only generates a new node when all nodes at shallower levels have been examined. Thereby it is guaranteed to find a shortest path if it exists. If each branch has *b* nodes, the number of nodes generated at level d will be b^d, at level d-1, b^{d-1} in general, the number of nodes generated by DFID is therefore

$b^d + 2b^{d-1} + 3b^{d-2} + ...db$ (Korf 1999) [20].

The space complexity on average case is $O(d)$, where d is the depth of the goal node. However the time complexity on average case is $O(b^d)$, where *b* is the branching factor and *d* is the depth.

The infinite deepening problem that DFS faces could be solved by limiting the depth of search with DFID, and also the memory problem BFS algorithm faces is eliminated by the DFID algorithm. The technique of limiting the search depth is called *Limited Iterative Deepening Depth First Search*; unfortunately limited DFID faces another

problem which is incompleteness. If the depth limit *d* is chosen with less than the depth where the goal occurs, then the algorithm returns no solution even though there may be a solution available deeper than the depth limit *d*.

4. Heuristic (Informed) Search

Judea Pearl (1984) defined heuristics in his famous book [33] by just that title as follows:

"Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be most effective in order to achieve some goal".

Some AI problems by their very nature require more time than others to find an exact solution, such as the traveling salesman problem (TSP), certain chess positions, etc. We call these kinds of problems NP hard problems. NP Problems in the realm of theoretical computer science are approached by approximation algorithms. AI approaches to these types of problems employ heuristics to try to ameliorate the combinatorial explosion. Still, in the worst case scenario, the time and space complexity of these algorithms is exponential.

The goal of heuristic search algorithms is to overcome the limitations that blind search algorithms face, such as the number of nodes generated and memory problems. For achieving this task heuristic search algorithms:

1. Decide which node to expand next, instead of performing expansions in a strictly breadth-first or depth-first style.
2. Decide which successor or successors to generate when generating nodes, rather than blindly generating all possible successors at once.
3. Decide that certain nodes should be discarded (or pruned) from the search tree. (Kopec, Cox, and Marsland [16])

Some well-known informed search algorithms are Hill Climbing, Best-First Search (A*), Iteratively Deepening Best-First Search (IDA*) [20], Recursive Best First Search, Beam Search, and Branch and Bound.

4.1. Hill Climbing Search

Hill-climbing [23, 35], is a search algorithm based on local optimizations. It is the first intelligent search algorithm. It is a greedy algorithm in that it has no ability to keep a history, and recover from mistakes. It simply builds a solution path by using heuristics or a strategy. It does not have a backing up feature. If the search is stuck to a dead end, there is no mechanism to backtrack and to search a different part of the solution domain. For example if you are looking for your parked car in a multi-level parking lot, if you are on the wrong level, even if you are in the right geographic area for your car, there is no way for you to find your car on this level. Let's say you parked in 5th level and location 553, but you are looking on the 6th level and the location 553 which doesn't exist. This problem is called ridge problem.

The disadvantage of the Hill Climbing algorithm is its "memory-less property". Because of that if it gets stuck in local maxima, and cannot backtrack.

4.2. The Best First Search

The problem with Hill Climbing is that its memory-less greedy feature often leads it to a local maximum or a dead end. It cannot recover from mistakes and cannot backtrack and try to search in a different part of the solution domain. The problems that Hill

Climbing faces lead us to a more intelligent search algorithm; The Best First Search [33] (see figure 4.2.1). It has the ability to keep track of the history and backtrack from local maxima or false leads to dead ends, resulting in search for a goal in different parts of the solution domain. A key component of Best First Search is its heuristic function h(n) . With the plain Best First Search, h(n) estimates the cost of the cheapest path from *n* to a goal node by employing some strategies or using some partial information. The estimated cheapest path is expanded in each iteration.

```
// Best First Search
BestFirstSearch(Root_Node, Goal)
{
Create Queue  Q
Insert Root_Node in to Q
While Q is not Empty
Do{
G= remove from  Q
If(G=goal) return solution path
          While (G has Children)
                    do{
                    if (child is not inside Q)
                    inset child node in to Q
                    else
                    insert child which has minimum value in to the Q
                    delete all the other nodes
                    }//end of second while
Sort Q by the value // Smallest node on the top
}// end of the first while
Return failure }
```

Figure 4.2.1. Pseudo-Code for the Best First Search

4.3. A Algorithm*

A* Algorithm [35] is a well-known form of the Best First Search. The A* search minimizes the total estimated solution cost by using an evaluation function f(n).

f(n)=g(n)+h(n).

g(n)→ is the actual cost of reaching a node n from a start node.

h(n→ is the heuristic estimation for getting from the node n to the goal.

*A** maintains two lists which are called the Open, and Closed List. The Open List is a priority list that contains the frontier nodes that will be expanded in the next iteration. Before a node is inserted in to the Open List, its weight is evaluated by a heuristic function *h(n)*. The difference between the estimated value and the real value is the error rate of *h(n).*The nodes in the Closed List are already expanded and removed from the Open List.

If the first element in the Open List is not a goal node, the algorithm generates its descendants. If an examined node is already in the Open or Closed List, the algorithm makes sure to keep the one which leads to the shortest path.

The A* algorithm stops searching if it meets the goal node or the Open List is empty. In the first case the algorithm returns the solution path leading to a goal, while in the second case the algorithm returns no solution path. If the A* algorithm employs an admissible heuristic function, then the A* algorithm will generate optimal solutions for problems which have a small state space search tree, but for problems which have a large search domain, the A* algorithm faces a memory problem.

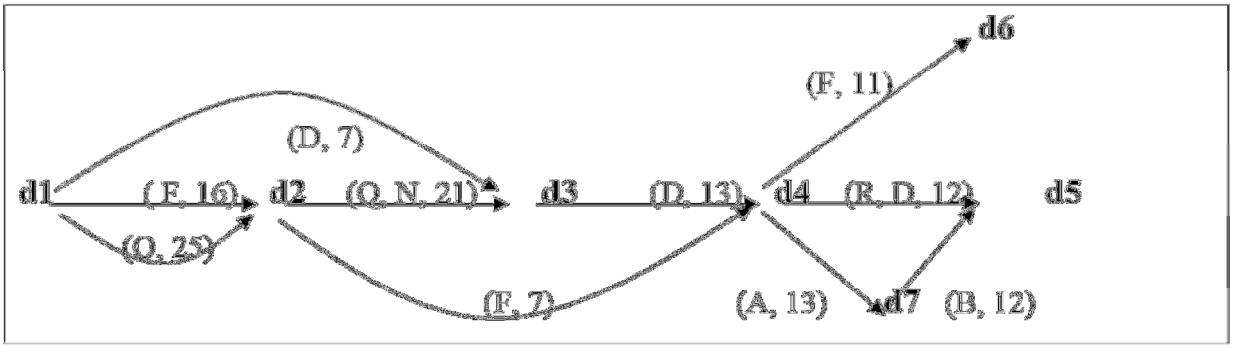


Figure 4.3.1. A state-space graph for a hypothetical subway system

The problem in figure 4.3.1 is an example of Best First Search presented by Kopec, Cox, and Marsland [16]. The search tree for this graph is shown in figure 4.3.2.

This example is a graph representation of a segment of the New York city subway system. The problem is to find the shortest path from d1 to d5 by taking different trains. A good heuristic for calculating the cost of traveling between stations can be defined by express or local trains; local trains cost more than express trains.

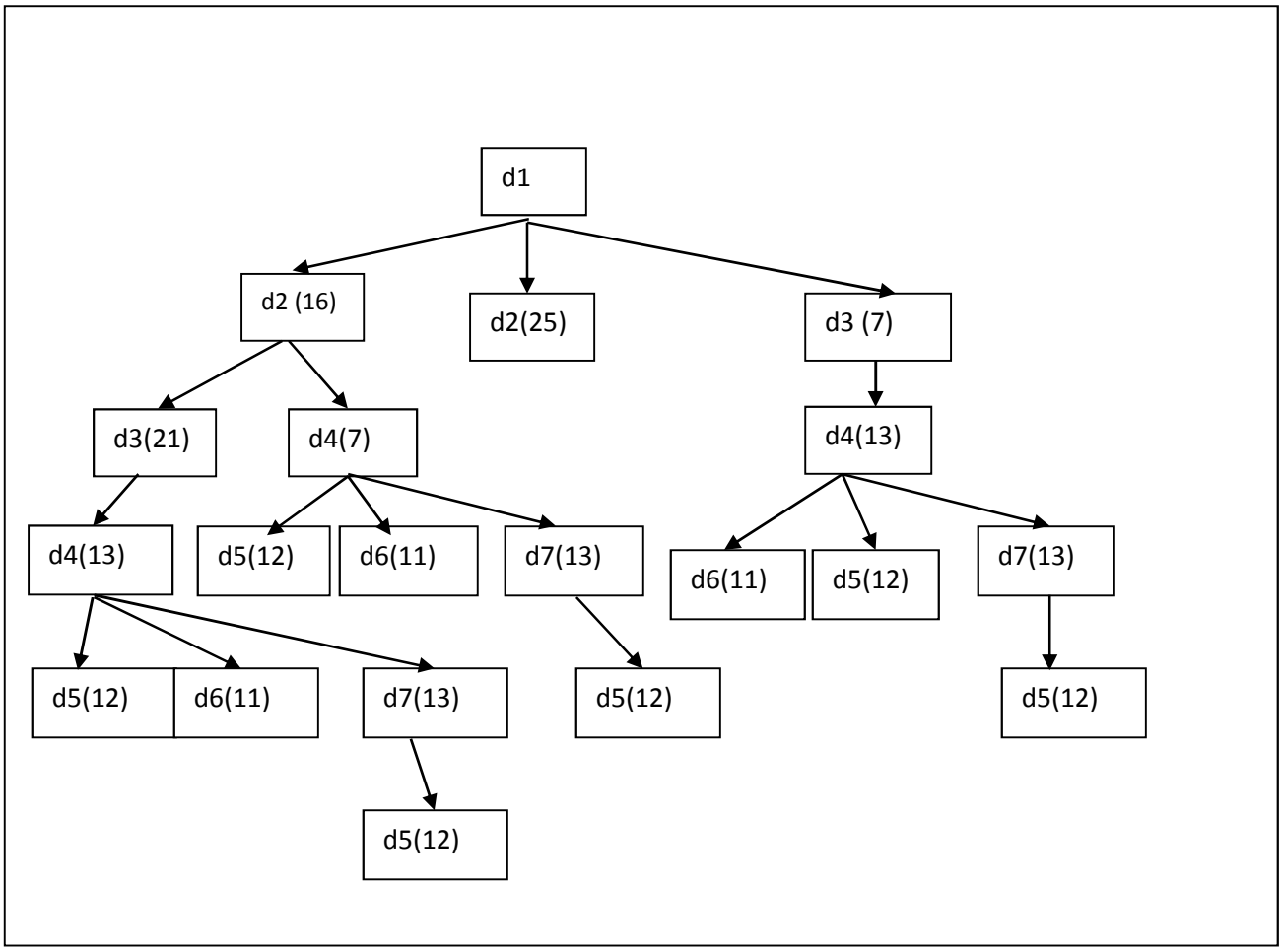


Figure 4.3.2. Search tree for the Graph in Figure 4.3.1.

If we follow the A* algorithm the shortest path will be taking the D train to d5 without transferring.

4.4. Iterative Deeping A (IDA*)*

Another instance of heuristic search algorithms is IDA* [20]. It is intended to deal with the memory problem that A* faces with complex problems. It uses the same heuristic function *f(n)* as A*. IDA* first performs a Depth First Search which keeps track of the cost, f(n)=g(n) + h(n) of each node generated. As soon as a node is generated whose cost exceeds a threshold for that iteration, its path is cutoff. The threshold is estimated

by a heuristic function. The main difference between IDA* and regular iterative deepening is that the deepening constant *d* for DFID algorithm is a variable ***x*** that starts from 1 and increasing one by one, such as, x = 1, 2, 3, 4, 5 …. n. Here iterative deepening is based on an evaluation function *f* rather than the regular depth d.

Since IDA* performs a series of depth-first searches, its memory requirement is leaner with respect to the maximum search depth. If the heuristic function h(n) is *admissible* [2], IDA* is optimal.

4.5. Recursive Best First Search (RBFS)

RBFS is a simple recursive version of BFS which tries to use less space. It is a Best First Search that runs in space that is linear with respect to the maximum search depth. RBFS generates fewer nodes than a regular BFS, and IDA*.

It keeps track of the heuristic estimated value f of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion backtracks to the alternative path. As the recursion backtracks, RBFS replaces the estimated value of each node along the path with the best f value of its children. RBFS is more efficient than IDA*, but still suffers from generating more nodes and eventually is faced with space problems.

4.6. The Beam Search

In the beam search [35] the investigation extents through the search tree level by level, but only the best *X* nodes are expanded. *X* is called the beam width. The algorithm

[2] A heuristic is admissible if it finds the shortest path to a goal node if it exists.

attempts to improve on the memory problem that BFS faces, by reducing the search space to a narrow beam width. BFS is used to build its tree by investigating all the nodes at each level without using any heuristic. Beam search keeps track of explored nodes in a priority list, and splits the list into two parts on the *x (x is the* index of the list), then pruning the part which exceeds the threshold criteria defined by the heuristic function. Then it investigates the nodes in the other part. This procedure persists until a solution has been found or the list is empty.

With an admissible heuristic function the beam search eliminates the space problems that all the other heuristic algorithms are faced with. The algorithm incurs some additional costs and is incomplete. With a small beam width the algorithm will work just like the DFS without backtracking. A solution could potentially be eliminated in some parts of the tree which have been pruned.

All heuristic search algorithms of this type have one characteristic in common: they each employ a heuristic function and based on the heuristic estimation they investigate the search tree according to their algorithmic steps.

There is another type of heuristic search algorithm which is referred to in the literature as “looking backwards”. Instead of looking forwards and taking forward steps according to a heuristic function, this class of algorithms looks backwards and takes further steps based on the distance from the start node. Branch and Bound is a well-known example of these algorithms that looks backwards.

4.7. Branch and Bound

The Branch and Bound (B & B) method was first developed by A.H Land and A. G Doig (1960) [22]. The B & B algorithm is mainly relevant to NP complete problems. The B & B algorithm has two parts: splitting a search tree and pruning some nodes of the tree. The first step of the algorithm is to split a solution domain into some smaller sub-domains; given a set *S*, divide S into two or more subsets such as S1, S2, S3, and S4. This procedure is called *branching*. The second part of the B & B algorithm is to prune some of these sets according to the heuristic function. This part of the B & B algorithm is called *bounding the solution domain*. The procedure goes on recursively until the solution tree has no branches to prune. At that point the algorithm stops.

The efficiency of the B & B algorithm is strongly dependent on efficiently splitting and pruning the solution tree. Hence the heuristic function chosen for splitting and pruning must be admissible. If an admissible heuristic function has been chosen, then B & B can generate optimal solutions.

5. Bidirectional Search Algorithms

The bidirectional search combines two searches simultaneously to achieve a task (see figure 5.1): One from a start node towards a goal node, and the other searches from a goal node towards a start node. If forward and backward searches expand a tree with branching factor, *b,* at level, *d,* then the complexity of each search is $O(b^{d/2})$ with a total complexity 2 x $O(b^{d/2})$ which is much better than the complexity of the unidirectional search $O(b^{d})$. Experimental results show that this is true only in theory. Usually, the two searches do not collide at a sub-goal (nodes *n* or *m* in figure 5.2 are two sub-goals). Figure 5.2 illustrates the essence of the *missile metaphor problem*. The forward search generates a solution path from *s* towards the left side of the solution domain, and the backward search generates a solution path from *g* towards the right side of the solution domain. The two solution paths do not intersect at the center of the solution domain. This missing frontier problem produces two A* searches instead of one: One search is from the start node *s* to the goal node *g,* and the other is from the goal node *g* to the start node *s*. The dark thick lines from figure 6.2 show solution paths from both sides; the forward solution path is P_1 = { s, t .. l, o, n .. y}, and the backward solution path is P_2 = {g, i .. k, m .. x}. The total time complexity is $2 \times O(b^{d})$.

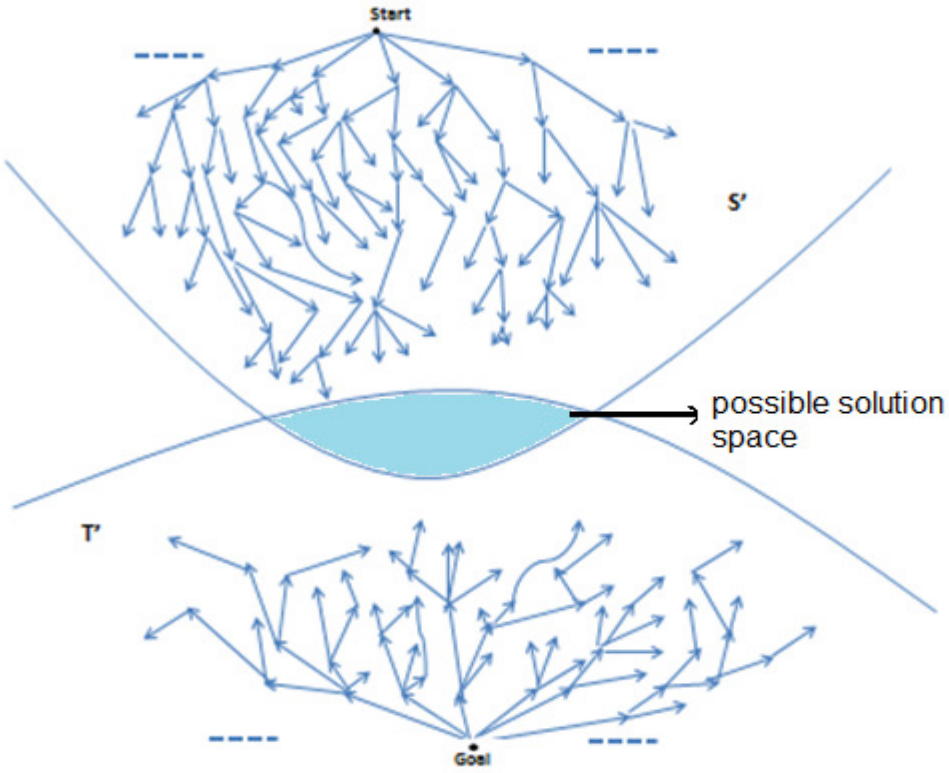


Figure 5.1. Bidirectional Search

Two main drawbacks of the traditional BDS were essentially responsible for the curtailment in research in this area until the early 1980s. The problems involved the following:

a.) The two frontiers may pass each other (known as the *missile metaphor*)

b.) If the two frontiers meet, much effort would have to be spent finding an optimal path (more computation to find an optimal path).

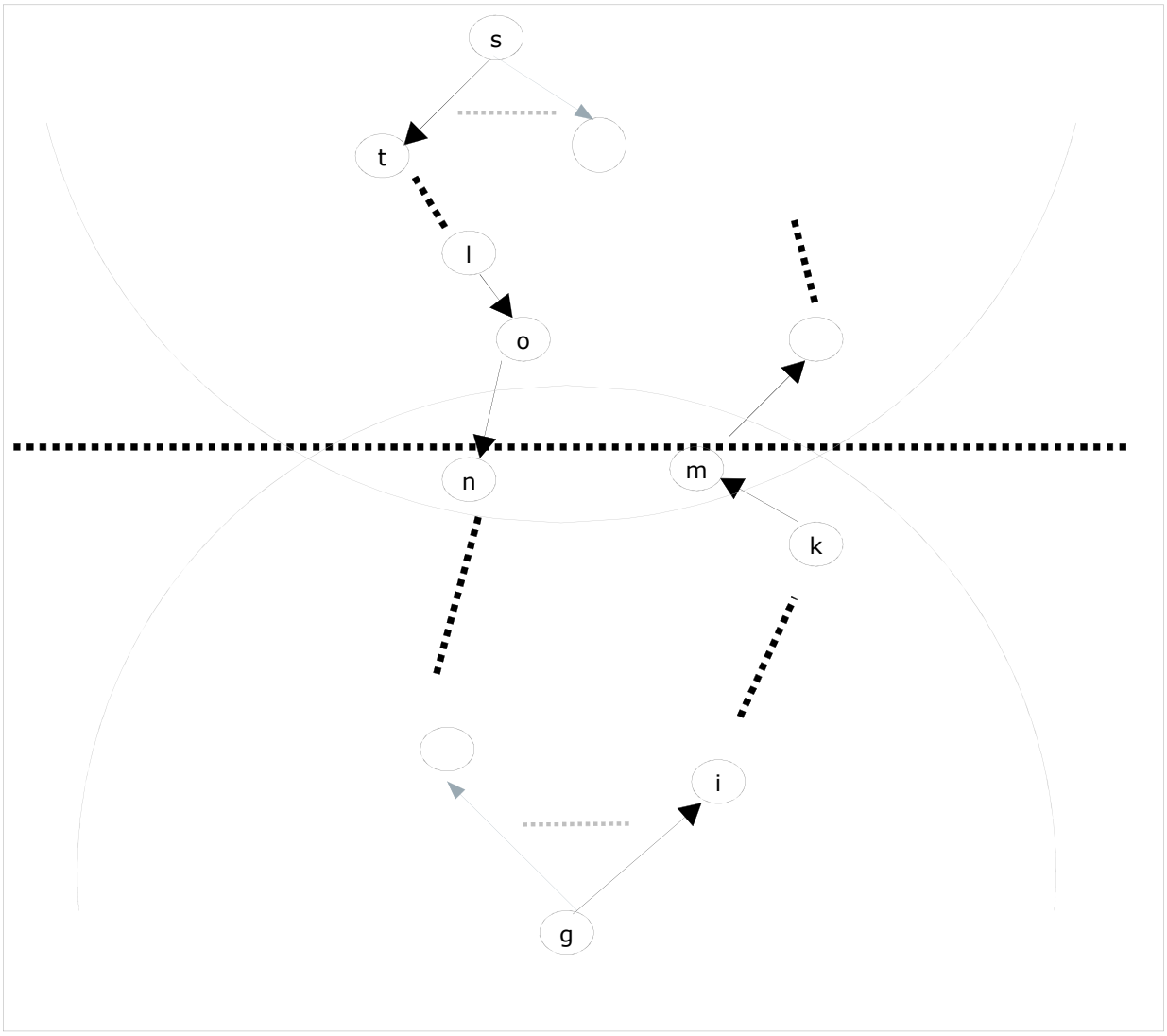


Figure 5.2. Missile Metaphor Problem with BDS.

The first problem was addressed by De Champeaux in 1983 [5] and Kwa in 1989 [16]. As we mentioned above, for a long time it was believed the search frontiers could pass each other. De Champeaux in 1983 [5] and Kwa in 1989 [16] addressed the missile problem. De Champeaux built a series of algorithms that used a technique called *wave-shaping*. Wave-shaping applies *front to front* evaluation instead of "*front to end"* evaluation. With *front to front* evaluation, instead of targeting a goal node (*front to end* evaluation), the heuristic targets the frontier of a search in the opposite direction. After

this improvement research on the bidirectional search resumed and newer algorithms were developed. Some of them are: BS*[15], Perimeter search [8, 9], generic approaches to BS [11], etc. The main aspect of all these approaches is to find a solution to a search problem within a small number of steps and generate as few nodes as possible. Besides the unidirectional and bidirectional search, newer research in AI has been focused in the area of pattern databases, dual search, and single frontier bidirectional search [9, 10, 17, 40, 41].

In the following sections we investigate a few traditional and non-traditional bidirectional search algorithms. The differences are that some of them generate a solution faster than others, and some generate a solution requiring fewer nodes than others. In addition to these methods, we also summarize the probabilistic methods.

5.1. Front-To End (BHPA)

Front-to-End Evaluation (BHPA) [34] approach is the first method developed by Pohl, it employs two A* searches; one is a forward search from start node to the goal node, and the other search is a backward search from the target node to the start node. The search frontiers switch directions by the following condition.

```
If (|OPEN1|≤|OPEN2|){
     d=1;
 } else{
  d=2; };
```

$OPEN_1$ and $OPEN_2$ are the lists employed by the searches from the start node toward the goal node and from the goal node toward the start node.

Let us call the search from start state to the goal state *S1*, and the search from goal node to the start node *S2*. The search direction will change from S1 to S2 if $d_1 > d_2$, where $d_1$is the depth of S1 and d_2 is the depth of S2. Whenever the frontiers collide, a solution is found. If the frontiers pass each other (which is the worst case scenario), the complexity is two A* searches. An improvement of this approach (BHFFA) published by De Champeaux (1977) [3], includes *the missile metaphor* problem.

Another improvement to these basic algorithms is BS* developed by Kwa (1989) [21]. BS* uses the same algorithm structure with some additional heuristics, such as *nipping,* and *pruning*. If a node selected for expansion and this node is already in the Closed List of the search in the opposite direction, it can safely added to the Closed List, without further expansion. This heuristic is called *nipping*. Removing all of its descendants from the Open List of search in the opposite direction is called *pruning*. Experimental results [13] show that a regular IDA* search is better than BS* for some classes of problems, such as Maze Problems. Experiments also show that BHPA could find a path faster that an IDA* search [13]. But there is no guarantee that the path is optimal. After a path has been found, the situation described by Lemma 1 (below) could occur. However an optimal path must be inside the set S'∩T'. Hence, a second search must be done in the S'∩T' (see Figure 5.3).

Lemma 1: the path from *p* to *m* is not optimal

Proof: there may be some nodes j, z and i such that $j, z, and\ i\ \in S' \cap T'$ which have a path from p->i->z->j->m than may be shorter than the path from p ->n->m. (see figure 5.1.1)

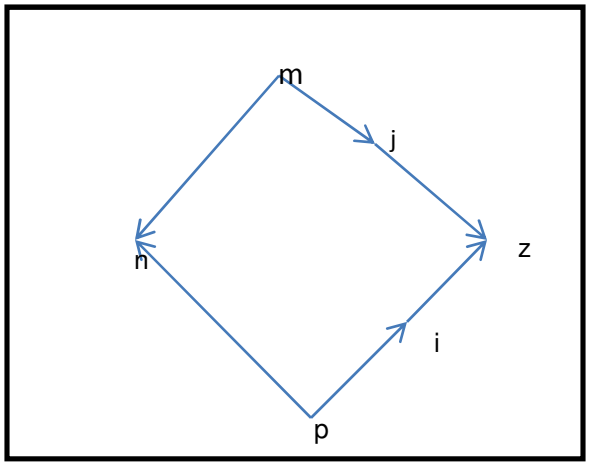


Figure 5.1.1. Optimality problem with BDS.

5.2. Front To Front Algorithm (BHFFA2)

Front-To-Front Evaluation (BHFFA2) [3, 4] approach is a successor of BHFFA. It consists of two loops; first loop is; "*find-a-path"* loop, and the second loop is; "*find-the-best-path"*. For f*ind-a-path* loop, the frontiers begin searching while S'∩T'=Ø [see Figure 6.2.1]. After the frontiers meet, S'∩T is no longer empty (S'∩T'≠Ø) [see figure 5.2.2], and a path has been found. After that, the algorithm switches to the second loop which is "*find - the-best-path"* [see figure 5.2.2]. A new search performs in S'∩T' until a best path found.

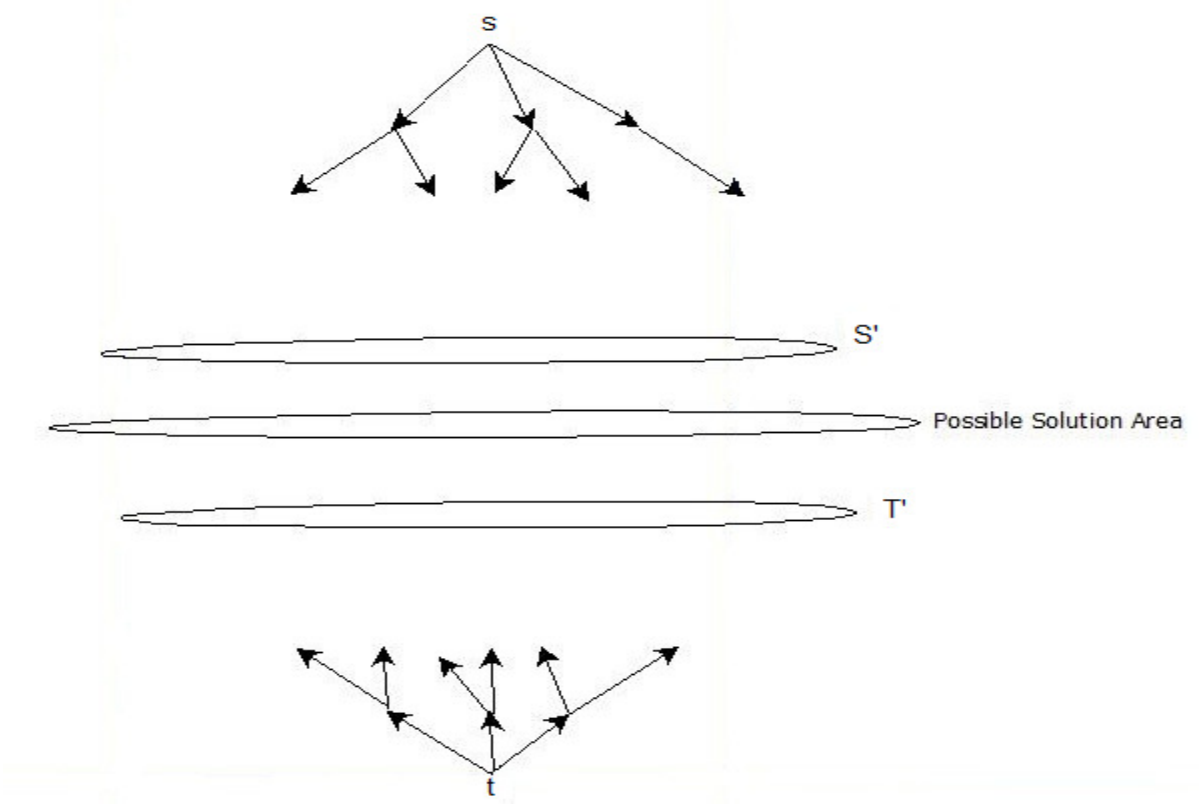


Figure 5.2.1. Before two search frontiers met. S' is the frontier nodes for search from start node, T' is the frontier nodes of search from goal towards start node.

In the worst case scenario, the frontiers will not meet at the center of the search space. In this case, a technique called "wave-shaping" is used to force the frontiers to meet. This technique eliminates the missile metaphor. In this situation the search space for "*find-the-best-path"* will be larger, and it will cost more time than a single A* search to find an optimal path. Some experimental results show that even BHFFA2 could find a path faster that a unidirectional search, although the number of nodes generated is greater. The running time is also higher than IDA* for the Maze problems**.**

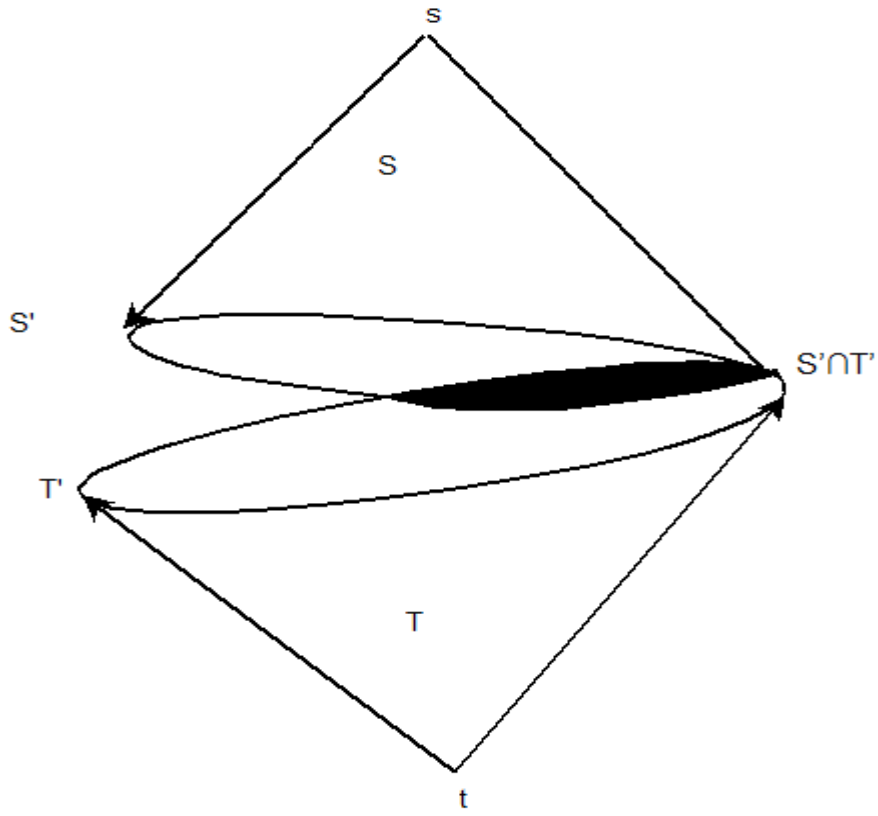


Figure 5.2.2. When the BHFFA is in the find-best path loop. S = the collection of nodes reached from start, T = the same with respect to goal, S' = the collection of nodes which are neither in S or P, but are direct successors of nodes in S or T. T' = the same with respect to T.

5.3. Perimeter Search (PS)*

This is the bidirectional version of IDA* and A*. This technique starts searching from one frontier up to a constant *m*. *m* is a fixed number and is called perimeter depth. The final frontiers of this search are called the *perimeters.* A second search starts from the opposite direction which targets the perimeters of an opposite search. Even the PS* has some advantages over the traditional bidirectional search, such as the number of nodes generated. Experiments shows that IDA* generates a solution faster than BIDA*

for the Maze Problem, but experimental results also show that BS* is better than IDA* for some classes of the Fifteen Puzzle Problems [13]. The main drawback of the PS* algorithm is that it spends too much time performing heuristic calculations. Dillenburg and Nelson [8] tried to eliminate this drawback, by calculating the heuristic differences between frontiers once. For each iteration, instead of computing the heuristic value for each node they just subtracted 1 from the current heuristic value of the examined node. Whenever the heuristic value of a node becomes lower than the minimum threshold, recalculate the heuristic value for these nodes. With this additional idea, the number of heuristic evaluations is reduced by 74% when solving 25 random Fifteen Puzzles [8]. Another drawback of the perimeter search is determining the perimeter depth. There is no mechanism for defining the optimal depth for parameter search.

To minimize the heuristic calculation problem with PS* algorithm, Manzini (1995) introduced a new perimeter search called “Improved parameter search” (BIDA*) [27, 28]. It uses a more efficient technique for pruning the non-optimal paths. Heuristic evaluation is one of the main drawbacks of perimeter search. BIDA* reduced the number of heuristic evaluations. The algorithm BIDA* explores a path until the heuristic value of the last node exceeds the current threshold. BIDA* is a perimeter search algorithm, and it generates exactly the same nodes as the IPS* does.

5.4. Generic Approach (GBS)

This approach was developed by Kaindi and Kainz [13] Instead of changing the search direction more than once as the traditional BDS algorithms do; it allows a one-time alteration of the search direction.

The steps of the algorithm are;

- Assign as much memory as possible to one frontier
- Perform a unidirectional search from that frontier
- If a solution is found, return the solution path.
- Otherwise perform a search in the reverse direction [13].

In the last step, the reverse search uses the database built up by the first search, and the available memory. This algorithm looks like the BIDA*, except that it switches frontiers just once and does not have a perimeter depth limit.

If the first search cannot find a solution, then a leaner search such as IDA* is initialized from the opposite direction towards the frontiers of the first search. The second search cannot terminate as soon as it finds a solution, because it might not be the optimal solution. A better solution may exist. The first solution is stored and used as a threshold for the rest of IDA* searches, whenever a path exceeds the threshold, its next iteration are cutoff.

5.5. Dynamically Improving Heuristic Approach (DBS)

This method was also developed by Kaindi and Kainz [3]. The algorithm adds a learning factor to the Generic Approach. This approach basically uses a learning method to improve its heuristic function *f(n)*. The heuristic function of the reverse direction search learns from the opposite direction's heuristic function f(n) and develops its own heuristic function *f'(n).* The advantage of this method over the regular Generic Approach is that

the error rate for the reverse side's search is lower. If the heuristic function is admissible, then the heuristic value *h* must be always less than the real cost *c* ($h \leq c$) the difference between *c* and *h* is the error rate of the heuristic function. This difference is used as the error rate of the heuristic function, and is added to the *h* value of the opposite search.

The experimental results posted by Kaindi and Kainz show that their algorithms are better than the traditional bidirectional search algorithms as well as the IDA*. For Maze problems, GBS and DBS perform better than IDA* and all the other bidirectional search techniques.

5.6. Divide and Conquer Bidirectional Search

This algorithm was presented by Korf [19]. The algorithm performs a BDS but it only saves the nodes of one side. It does not save the nodes of other side. Whenever two frontiers meet only one part of the solution path has been found. The next iteration of the algorithm uses frontier node as a start and repeats the same procedure. This recursive procedure keep repeating until all nodes of the solution path are found. See figure 5.6.1. the advantage of this algorithm is that it reduces the memory requirement from $O(b^d)$ to $O(b^{d-1})$

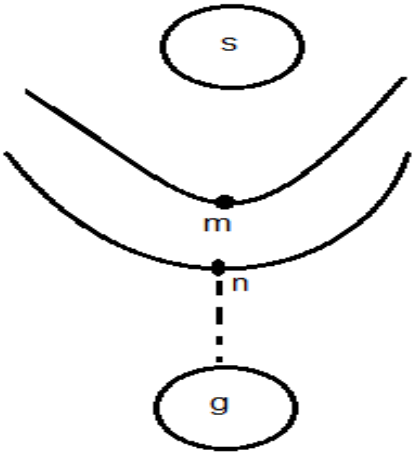


Figure 5.6.1. Dive and Conquer Bidirectional Search

5.7. Single-Frontier Bidirectional Search (SFBD)

Recently, Felner, Mondenhauer, Sturtevant, and Schaeffer [4] introduced the Single Frontier Bidirectional Search (SFBS). Unlike traditional BDS, SFBS keeps track of single frontiers. In SFBS nodes are defined as a pair of states, such as state *s* and *g,* denoted by node N*(s, g)*. SFBS generalizes the dual search (introduced in the following section) to all possible state space problems. At a particular node, the SFBDS decides to search from a start node *s* to a goal node *g* or from a goal node *g* to a start node *s*. The heuristic function *f* computes the heuristic differences between two states. In the next iteration the node which has the smallest *f* value is expanded. Finding the shortest path between *s* and *g* can be solved via recursion. For example if we want to find the shortest path between state *s* and *g* (see figure 5.7.1), the root of the search tree (represented by *s* and *g)* will be expanded. If s is expanded the possible nodes generated are*: (a, g)* and *(b, g)*. If *g* is expanded then the possible nodes generated are: *(s, n)*, and *(s, m)*. A jumping policy decides which state to expand *s* or *g*. The node with

minimum f value will be expanded. Within the chosen node N, the choice of expanding s or *g* for the next iteration depends on the jumping policy. The search proceeds until the *f* value between two states is 0 or the two states are same (s = g). The jumping policies employed to choose which state to expand in the next iteration depends on the problem domain. For example with Fifteen Puzzles, the state which has the minimum branching factor expands.

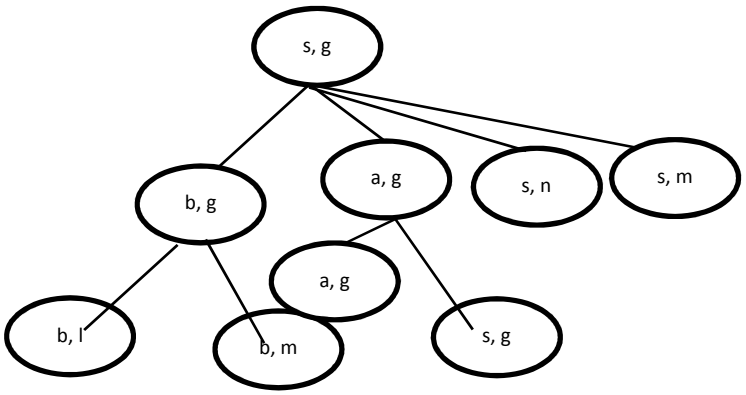


Figure 5.7.1. Single Frontier Bidirectional Search

5.8. Pattern Databases

The Manhattan Distance Heuristic is one of the main heuristic evaluation function used for the *single agent path finding problems*. Alternatively, we can derive the Manhattan distance by observing that: to solve certain AI problems we first must reach some well-defined sub-goals; from there we can reach a desirable goal. For example, sliding-block puzzles contain sub-problems which involve getting some blocks to their goal locations or temporary locations. Before the whole puzzle can be solved, the sub-goal must be reached. The sub-goals are a lower bound on the actual solution of the puzzle. This

new idea suggests a new heuristic evaluation function for *single agent path finding problems*. A *pattern* is a sub-goal of a problem (see figure 5.8.1). For example, a pattern for sliding block puzzles is: if some tiles are in their final locations and some others are not in their final locations, then this state of the puzzle is called a pattern. Blocks which are in their final locations are called fringe tiles (see figure 5.8.1). Blocks 3, 7, 11, 12, 13, 1, 4, 15 are fringe tiles. These fringe tiles can be on any location on the board. The number of possible permutations of fringe tiles is 518,918,400 [5]. The database of these paths is called a *pattern database*. If each path is stored in one byte, then all of the pattern databases can be stored in 500 megabytes of memory [5]. After a pattern database has been created the solution to a Fifteen Puzzle is to just the solve an Eight Puzzle problem. The search space is reduced from the size of the Fifteen Puzzle to the size of the eight puzzle spaces. The database created by identifying certain tiles in their location is called a fringe pattern [5].

The original non-additive pattern databases were first introduced by Schaeffer and Culberson [5]. They couldn't solve twenty-four puzzles due to the large number of entrees in the database. An alternative to non-additive pattern databases is disjoint pattern *databases*, introduced by Korf in 2002 [17]. The basic idea of disjoint pattern databases is to partition the pieces of a problem into disjoint groups, and to solve each group independently by creating pattern databases.

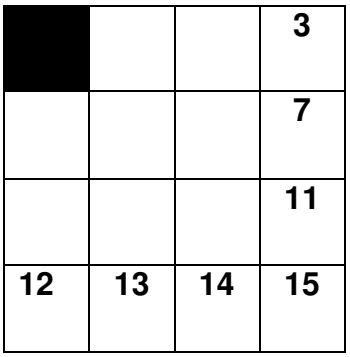


Figure 5.8.1. Fringe Tiles in a Pattern Database.

5.9. Dual Search

Some properties of search problems such as duality commonly explore opportunities to improve the efficiency of search algorithms (see figure 5.9.1). The states of some problems such as sliding-tile puzzles have dual states: that is both states (regular and dual) share some properties such as the distance from a goal node or the distance from a start node. Dual Search [40, 41] employs any heuristic search algorithm to find a goal. Whenever it reaches a node *x*, if heuristic estimator h(x) of x is smaller than its dual node's heuristic estimator $h(x^d)$, then it switches the search side from regular state to the dual state. This improves the chances to finding a goal state in less time and with fewer nodes generated. Comparing the heuristic estimator is one of the policies employed by the dual search for switching sides.

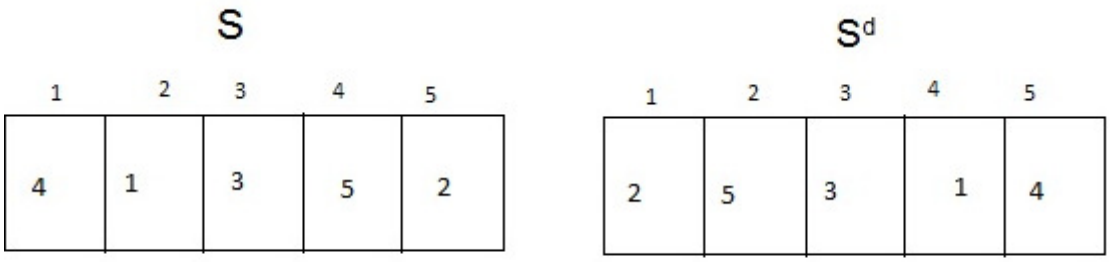


Figure 5.9.1. Regular State S and its Dual State S^d.

Dual search applies to two problem types: problems where the solution domain is a permutation state space *and* problems where the solution domain is a strictly permutation state *space*. Permutation state spaces consist of a set of m objects and n locations. The state space of such problems consists of different ways to place objects into the locations. If n = m (the number of objects is same as the number of locations) then the state space is called a strict permutation *state space.* Some well-known examples of these problem types are: Fifteen Puzzle, Rubik's cube[3] . The dual search is limited to these kinds of problems.

[3] Rubik's cube was invented by Erno Rubik of Hungary in 1975

6. Probabilistic Methods

Probabilistic methods [26, 35] such as simulated annealing, can also be applied to the searches of BDS. Probabilistic algorithms are non-deterministic; whenever the heuristic estimation of an examined node is acceptable, the node will be chosen for expansion, while the rest of the nodes will be discarded. However, regular search algorithms need to examine all the generated nodes. Probabilistic algorithms must have some kind of termination condition(s). Whenever those condition(s) are satisfied, the algorithm will not examine the remaining nodes generated and will switch to the next level. The rule for moving from one level to the next level is probabilistic. The conditions for accepting or rejecting an examined node are not static. For example, simulated annealing methods accept or reject an examined node by a probability, *p.*

$$p = \frac{1}{1 + e^{\frac{h(x)-h(y)}{T}}}$$

Eq.1: h(x) -> heuristic estimator of node x, h(y) -> heuristic estimator of node y, and T is a variable.

7. AI Problem Domains

As we have described above, AI problems are the most challenging problems, in most cases requiring exponential time to find an optimal solution. The challenges of AI search problems often involve: complex computation, more space, representation methods, and time. Hence, heuristic evaluation functions and diverse programing paradigms play an important role in the development of solutions. The solutions to AI problems require that most time is spent in finding an admissible heuristic evaluation function to solve them.

If we classify AI problems by their difficulty, the game of *tic-tac-toe* might be the least challenging, and chess might be deemed one of the hardest AI problems. There are two aspects to the difficulty of an AI problem [16, 17]: decision complexity, the difficulties encountered in trying to make a correct decision, and space complexity, the size of the search space. Based on these criteria, Checkers with a search space of more than 5 x 10^{20} [37] possible positions is a more complex example for our research. However, we believe that The Klotski class of puzzles such as Donkey Puzzle (described in the next sections) could be a good starting point for our research, and then as a second problem we have investigated the class of Fifteen Puzzles. Their solution space is less than that of checkers, and larger than the Klotski Puzzle. They are also a standard domain used in the AI literature to measure the efficiency and effectiveness of heuristic search techniques.

The time required to solve AI search problems and their corresponding memory requirements for storing a solution is crucial. Hence most search algorithms have been primarily analyzed by focusing on these two factors.

AI search algorithms are useful for solving *single-agent path finding problems, two-player games, and constraint-satisfaction problems.* Single-agent path finding problems are for finding the number of legal moves to convert a start state of a problem to a goal state. Some classical examples are sliding-block puzzles, and the travelling salesman problem. Two player games are played between two opponents, each player taking his/her turn to make a legal move. Some classical examples are Chess, Checkers, and Othello. The third class of problems is Constraint-Satisfaction Problems (CSP) whose characteristic is to map a given input to a desirable output. Some examples of CST are the eight queen problem (placing eight queens on a 8 x 8 chessboard) and scheduling-related problems.

If the problem ***P*** is a constraint satisfaction problem, the technique to solve problem ***P*** is to map its start configuration to the goal configurations by reaching some sub goal. Given the sub-goals of problem ***P*** are P_1, P_2, and P_3, to solve problem ***P***, we must solve P_1, P_2, and P_3. To solve P_1, P_2, and P_3, we must solve their sub-problems, after we solve all the sub goals we can obtain the final solution by recursively combining the results.

Path finding or the single-agent path finding problems need more sophisticated solution methods. Their sub-goals are not as simple to define, as those for constrain-satisfaction problems. Schaeffer [5] built Pattern Databases for solving the Fifteen and Twenty-four

puzzles, the Pattern Databases are databases of sub-goals of these puzzles. In term of reaching a goal position, some sub-goal configurations must be reached. Then from this sub goal the final goal configuration can be easily reached. This strategy is not generic, it is specifically built for the Fifteen and Twenty Four Puzzles, and some version for Pattern Databases (built by Korf [17]) can be applied to Rubik's cube.

The solution to path finding problems involves three mechanisms: a good representation method, a search method, and a heuristic evaluation function. A representation method is needed for representing the problem states; the heuristic evaluation function is needed for evaluating the heuristic values of nodes, and a search mechanism is needed for how to examine and evaluate the nodes, and in which order to store the generated nodes.

7.1. Klotski Puzzles

The Klotski (Donkey) Puzzle is one of several sliding block puzzles in which 10 wooden pieces of different style are placed on a 5 x 4 board, with two locations empty. The objective is to reach a state where the 2 x 2 piece is in a desirable location. For most cases the objective is to move the 2 x 2 piece to the center of rows 3, and 4 and columns 1, and 2. The puzzle consists of four 1 x 1 blocks, five 1 x 2 blocks, and one 2 x 2 block. See Figure 7.1.

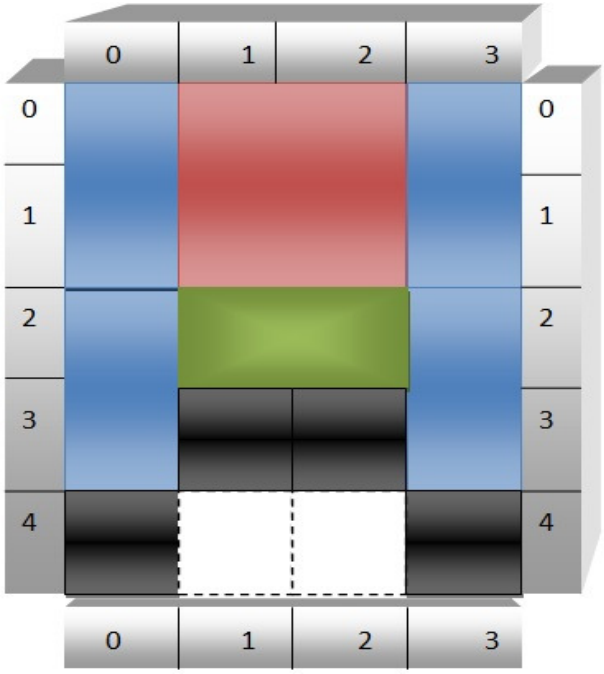


Figure 7.1. Starting Board Configuration of Klotski Puzzle.

We refer the blocks by labels such as 2 x 2, 2 x 1, 1 x 2, 1 x 1, and empty locations will be referred to by the number 0.

7.2. Fifteen Puzzle

Sliding-tile puzzles in general consist of n × n frame holding n^2-1 tiles and blank tile labels with 0. The Fifteen Puzzle a form of sliding-tile puzzles with n fixed to 4. It consists of 15 tiles labeled from 1 to 15 and the blank tile labeled with 0. To play the game you must move the blank tile to Up, Down, Left, and Right. Any tile that is horizontally or vertically adjacent to the blank may move into the blank position. Any such operation is a legal move. Each legal move creates a new problem state. A commonly used heuristic function to solve sliding-tile problems is the Manhattan Distance Heuristic (described in the following section).

The objective of this game is to position the numbers from 1 to 15 in ascending order where the numbers in the grid are initially in random order.

8. State Space search and the Cost Function

A state space consists of a set of states and a set of operators. The states are representations of problems and the operators are the legal moves that transfer problem states into other states. With our implementations we have added costs to the states of our state space search. The costs of states are used for defining which state to expand next. Unfortunately it is not possible to define a general cost function for all the problems. The cost functions are unique to the problem domains. For our implementation of Klotski Puzzle and Fifteen Puzzle we have employed the Manhattan Distance Heuristic and we also added the Manhattan pair distance heuristic as well. The estimated cost of a node N, $f(x)$, is the addition of actual cost from start node to N and the an estimated cost from N to the goal node.

$f(x) = c(s,n) + h(n,g)$, where $c(s,n)$, is the actual cost from node s to the node n, and $h(n,g)$, is the heuristic estimation from node n to node g.

9. New Types of Search Algorithms

A variety of AI search algorithms have been described in the previous sections. Some of them are considered to be amongst the main AI search algorithms such as BFS, DFS, A*, etc. and others use these main algorithms as a basis and include more heuristics to generate newer algorithms. For example DFID uses the idea of DFS and add a depth limit to the search. The depth limit increases by one for each iteration. IDA* combines DFID with A* and uses heuristic evaluation as a threshold instead of search depth.

The research we have done introduces two new AI search algorithms that eliminate the Closed List employed by most of search algorithms. We call them: the indexed Breadth First Search (IBFS) and Indexed A* (IA*).

9.1. Indexed Breadth First Search

9.1.1. Introduction

Various forms of the Breadth First Search (BFS) algorithm have been presented in the literature of AI search algorithms. The basic form of BFS algorithm employs two lists; one called Open List which is required to track the frontiers of the search tree and the other one is called the Closed List, and is required to store nodes which have been explored but do not lead to a solution(see section 3.2 for more details). A node, *N,* in the regular BFS is characterized by a 3 - tuple *{S, p, depth}*, where *S* is the state of a problem, *p* is the parent of *N*, and *depth* is the distance from start node to N.

If the possible node generation operators are well defined for some problem domain then we can build a strategy which prevents duplication. For example, for sliding puzzle problems such as The Fifteen Puzzle, a node may be generated as a result of a Left, Right, Up or Down move. If we could track this we can prevent possible duplications and cycles that may loop back from two levels down back to ancestors (see figure 9.1.2.1). The regular BFS does not track the information about how nodes are generated, and node generation is not labeled.

The policy of generating nodes without keeping track of how nodes are generated may create cycles. For example if the child of a *Left* move is a *Right* move, this will loop back to the parent with a *Left* move (see figure 9.1.2.2). The search tree for the basic form of BFS is an undirected graph. However BFS eliminates cycles and duplications by searching through the Closed List for the newly generated node. If the newly generated node is in the Closed List then the node will be discarded without adding it to the Open List, otherwise it will be added to the end of Open List. Some forms of BFS, check for duplicate nodes before including a node to the Open List. This mechanism adds a lot of search overhead when the search tree becomes larger. The second purpose of the Closed List is to build the solution path after a solution has been found.

If we have developed an algorithm to eliminate the following two problems then we can safely eliminate the Closed List employed by BFS.

- How to prevent duplications or cycles
- How to build the solution path after a goal node has been reached

9.1.2. IBFS

Unlike traditional BFS, IBFS has the mechanism to track how a node generated (see definition 1) and the Index of a state (see definition 2). These two parameters guide the search to avoid cycles and build the solution path easily. The contribution of IBFS is to handle the two issues we have stated above and also to save some space by completely eliminating the Closed List. Our experimental results show that IBFS algorithm is slightly faster than BFS (see table 12.5.2).

Another advantage of the IBFS algorithm is that the solution path produced is simpler. It does not need much computation. If you want to get the solution path from a goal node to the start node you need to follow the following three steps. If you just need a solution path from the start node to the goal node you just need to convert the index of the goal node to the base *b*, where *b* is the branching factor.

1. Convert the index of goal node to the base of maximum branching factor.
2. Switching each “Move Way” (described in the definition 1) to its dual move
3. Reversing the resulting string

How to define the “Move Way” and their dual moves [4] depends on the implementation and on the problem domain. “Move Way” and their dual for Fifteen Puzzle are shown in the Table 9 1.2.1.

[4] A dual move is the opposite move of a move. For Fifteen Puzzle, right move’s dual is left move, up move’s dual is the down move vies versa.

Move Way	Left	Right	Down	Up
Numerical Representation	0	1	2	3
Dual Move	1	0	3	2
Dual Move Way	Right	Left	Up	Down

Table 9 1.2.1. Parent Move and its Dual Move

For a node in the index 1567, the solution path can be obtained by the following simple three steps;

Step 1. $(1567)_4$= 120133

Step 2. Dual of (120133)=021022

Step 3. Reverse (021022)=220120.

If the maximum branching factor of a problem is known, then we can define the possible number of maximum moves when we expand a node. For a problem *P* with branching factor *b*, the maximum number of children of N is x, where x ≤ b. For example, the branching factor of the Fifteen Puzzle is 4 (Left, Right, Up and Down), so the number of nodes you could generate by expending a node is 4 or less.

Definition 1:

The *Move Way* is how a node is generated from a parent node. For example In figure 10.2.1.2, the node *a* is a result of a "1" or right move so the "M*ove Way"* for node *a* is "1". The "*Move Way"* for node *b* is "0", the "*Move Way"* for node *c* is "1", etc. The "Move

Way" of all the nodes must be a combination of legal moves (see figure 9.2.1.2). A parent node can have a maximum of *b* nodes.

Definition 2:

The index of a node is the location of a node in the search list. See figure 9.1.2.0. We can calculate the index of a node N by the following formula.

$i=4\times j+w$.

i → the move index of new generated node

J → the index of parent of N

w → the move way that generate node N.

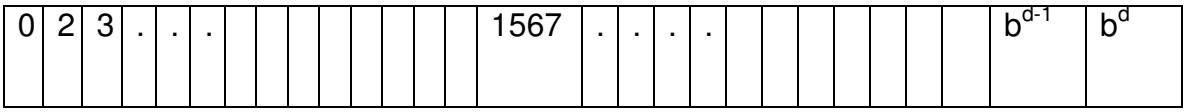


Figure 9.1.2.0. Index of Nodes

The structure of a node N in IBFS is characterized by the 3-tuple {S, MoveWay, Index}, where:

- S is the state of the node
- *Move Way* is how the node generated
- Index is the location of node in the Open List (the index of node in the Open List)

We don't keep track of the parent node. We label the *MoveWays* with numbers instead of strings, such as for the Fifteen Puzzle the possible moves are {0, 1, 2, 3}. We can make the following assignment for a "Move Way" of the Fifteen Puzzle; {left==0,

right==1, down==2, up==3} . If the branching factor of a problem is six then the possible "*Move Way"* is {0, 1, 2, 3, 4, 5}. Each generated state must be labeled with any of these "Move Way".

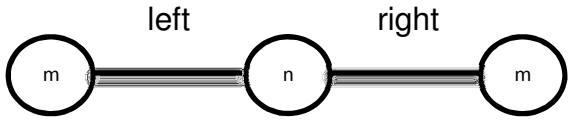


Figure 9.1.2.1. Prevent looping back to parent node

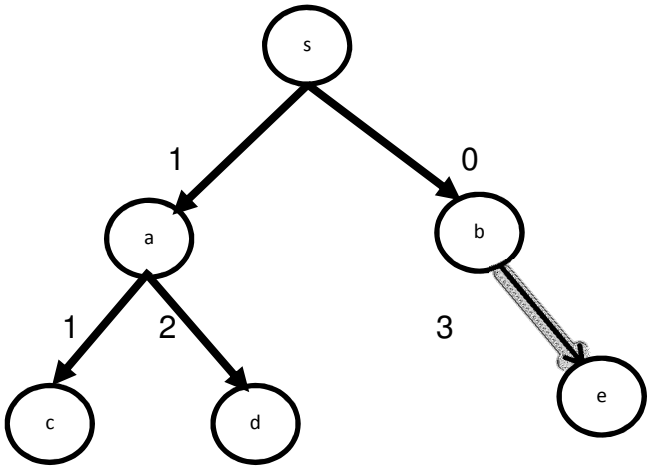


Figure 9.1.2.2. BFS Search tree representation of moves is 0 = "left" 1 ="right" 2 ="down" and 3 = "up" for the Closed List. [a,b,s] and the Open List is [c,d,e] .

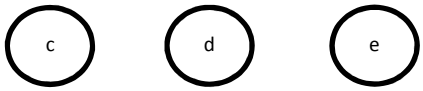


Figure 9.1.2.3. The IBFS search List. *Frontier* list is {c,d,e}.

If the solution domain of a problem is a map from states to the indices of states, then each state in this map has at least a coordinate or index. We claim if we know the index of a goal node then we can generate the solution path as we have described above. With IBFS we have redefined the task of search algorithms; the new track is to find the index of a solution configuration on the map. Once we know the location of goal state in the map, then the problem solved by converting the index of goal state to the base of branching factor of search tree. This policy will take care of building the solution path after a goal node is reached.

The purpose of retaining “Move Way” is to prevent first level duplication and the possibility of cycles that we have described above. But it does not prevent cycles that might occur at deeper levels. In the Fifteen Puzzle example the loops may occur at depths 1,6,7,8 and so on. If we search for the duplicated nodes before adding them to the Open List, then duplicate nodes are prevented, but it won’t prevent cycles. To avoid cycles, we employed the following strategy: If a node is duplicated, we set a parameter for this node. We called the new parameter the “Duplication Parameter”. The purpose of this parameter is to check the duplicated node’s “Move Way”. If a node N is already in the Open List, then its duplication N’ is generated. The N’ won’t be added to the Open List because N with the same state is already in the Open List. To avoid N from looping back to the root of the search tree, we set the “Duplication Parameter” for N to the dual “Move Way” of N’. So that, in the next iteration when we expand N, it won’t generate the parent of N’. In this way we can avoid possible cycles. See Figure 9.1.2.4

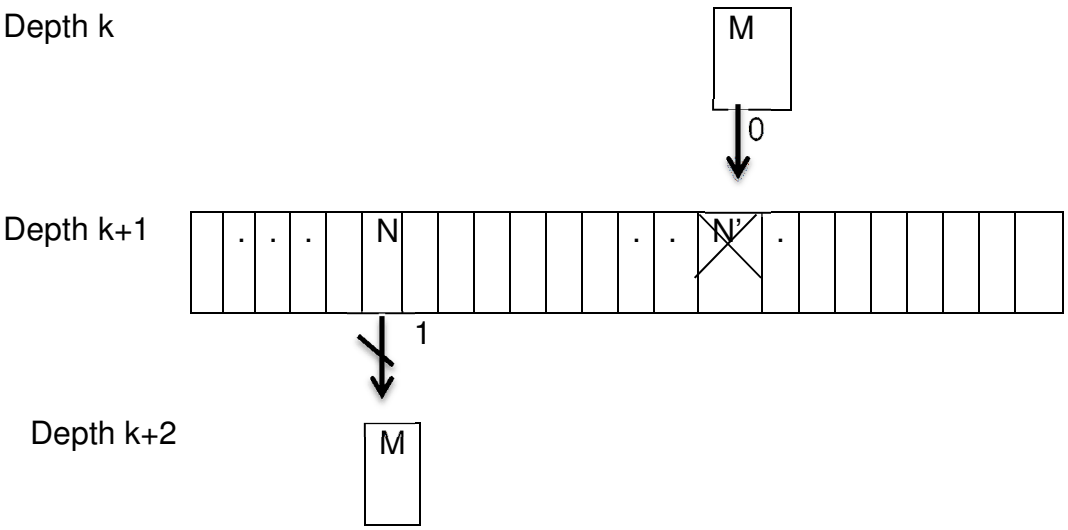


Figure 9.1.2.4. Check Duplicate Nodes and Prevent Cycles

9.1.3. Algorithm Components

IBFS uses only one list to track the frontier nodes and the parameters we have described above to avoid cycles while building the solution path. We called this list the Frontier List. We don't save explored nodes (see figure 9.1.3.1)

In addition to these main data types, depending on the implementation, you can add some other data types as well. For one of our implementations we keep track of the depth of nodes and for preventing cycle we keep track of another parameter (see above section).

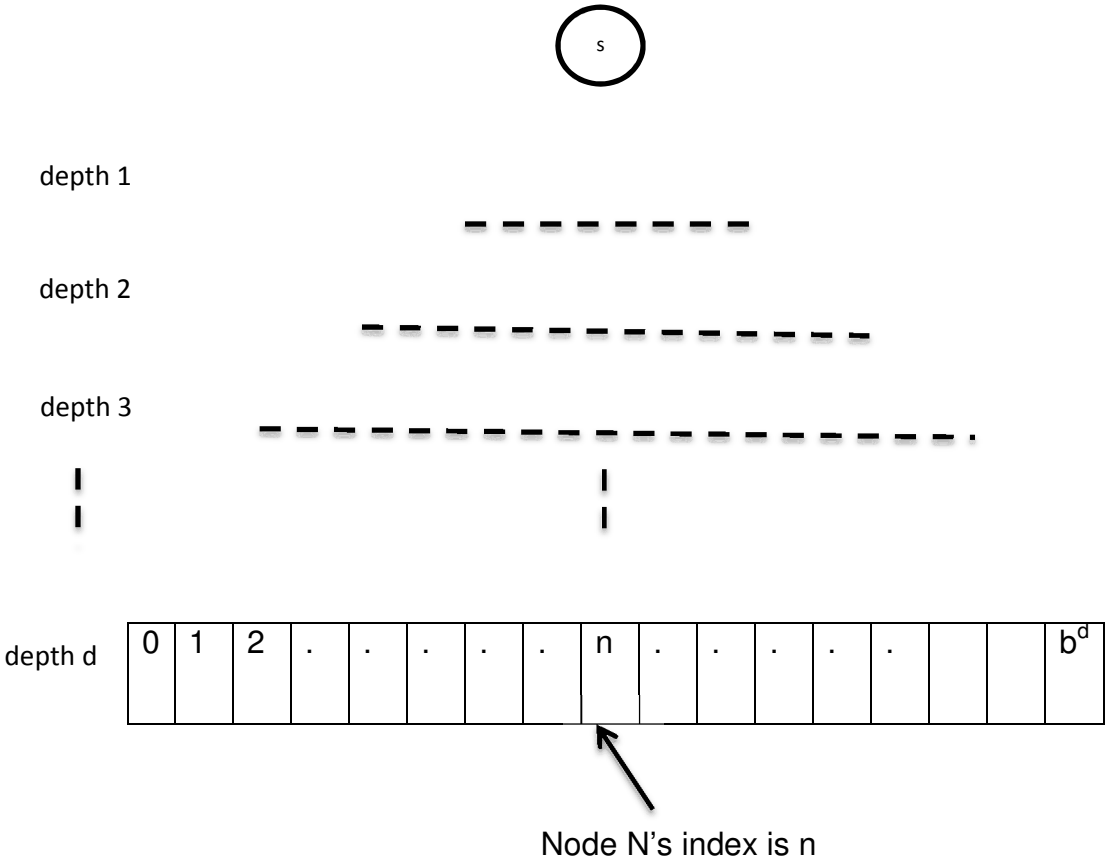


Figure 9.1.3.1. Find the Index of a Node.

The IBFS algorithm works as follows. First, the start node is expanded, and the parameters for each node are calculated. Before new nodes are added to the Frontier List, we check for any duplication. If there is no duplication and the new node's state is not a goal, then we add the new node to the end of Frontier List. In the next iteration the first element from Frontier List is examined. The procedure proceeds until a goal node is reached or the Frontier List is empty.

Algorithm:

// Indexed Breadth First Search

1. IBFS(Node S, Node G) // S is the start State and G is the Goal State

2. Add S to the Frontier List L

3. While L is not empty

3.1. Node N= First Element of L (Remove first element from L)

3.2. Expand (Node N, List L)

// expand the examined Node N

Expand (Node N, List L)

1. While N has Children

1.1. Create the State of all children

1.2. Calculate the parameters

1.3. If the new generated node X is a duplicated node, discard it

else

1.3.1. Check Solution(Node X) // if N' is the goal node then the procedure will terminate

1.3.2. add node to the end of L

9.1.4. Analysis and Complexity of the Algorithm IBFS

In this section we analyze the algorithm IBFS and compare it with the regular BFS. In the previous section we have shown that IBFS differs from BFS in that it uses two parameters to eliminate the Closed List employed by BFS. The algorithm IBFS does not track the Closed List as the BFS does. This policy saves memory and time. By tracking the index of frontiers, the IBFS algorithm eliminates two reasons for using the

Closed List. Now we theoretically analyze the benefits of eliminating the Closed List. As a result of the BFS and tracking, the index of nodes is generated.

Lemma 9.1.4.1. Suppose saving a node's cost is *x* space and *t* time, then algorithm IBFS will use less space and time to generate a solution.

For a problem with branching factor *b*, at depth (*d*-1) the number of nodes generated is:

$b^0 + b^1 + b^2 \ldots b^{d-2} + b^{d-1} = x$

At depth *d* the number of nodes generated is

$b^0 + b^1 + b^2 \ldots b^{d-1} + b^d = x + b^d$

The size of the Closed List for BFS is x, if we employed IBFS, we need b^d space to save the Frontier List, we will save *x* amount of memory.

Saving nodes and searching through the Closed List and the Open List for duplicated nodes, is a time consuming task. With IBFS we eliminate the time needed to search the Closed List and also by tracking the "Move Way" parameter we avoid certain levels of loops. This approach was published by Korf in 1984 [20].

At depth 0 we need to search for 1 node

At depth 1 we need to search b^1 nodes

at depth 2 we need to search $b^0 + b^1 + b^2$ nodes

At depth 3 we need to search $b^0 + b^1 + b^2 + b^3$ nodes

At depth n we need to search through $b^0 + b^1 + b^2 \ldots . b^{n-1} + b^n$ nodes .

Let $b^0 + b^1 + b^2 \ldots . b^{n-1} + b^n = m$, where m is the number nodes at depth n.

Let *t* be the amount of CPU time necessary per node search, searching *m* nodes will cost us $t \times m$ time. Hence Eliminating the Closed List will save us time as well.

9.2. Indexed A* Search (IA*)

With the regular A* search an admissible heuristic function enables nodes to be expanded. At each iteration the node with minimum evaluation value is expanded. The nodes which are expanded and do not lead to a solution path move to a list called the Closed List, and the newly generated nodes move to a list called Open List. The purpose of the Closed List is to avoid duplicated nodes and to build the solution path after a goal node is reached.

If we employ the policy of IBFS instead of BFS we can eliminate the Closed List employed by the A* algorithm. We call the new approach the Indexed A* (IA). The difference between IBFS and IA* is that, with IBFS the nodes are examined on a First In Last Out manner (FILO), but with IA* we save and expand nodes by evaluating their heuristic values, we always first examine the node which is closer to a goal node.

Algorithm:

// Indexed Breadth First Search

4. *IA* (Node S, Node G) // S is the start State and G is the Goal State*
5. *Calculate S Heuristic Value*
6. *Add S to the Frontier List L*
7. *While L is not empty*

7.1. Node N= First Element of L (Remove first element from L)

7.2. Expend (Node N, List L)

// expand the examined Node N

Expand (Node N, List L)

2. While N has Children

2.1. N' is new created children

2.2. Create the State for N'

2.3. Calculate the parameters For N'

2.4. Calculate Heuristic Value for N'

2.5. If N' is a duplicated node, discard it

else

2.5.1. Check Solution(Node N') // if N' is the goal node then the procedure will terminate

2.5.2. add N' to the list L by its heuristic value h // if h is the minimum value N' must be at the front of list, if h is the maximum value among the all heuristic values, then n' must go to the end of list L

10. Programming Language Paradigms and Their impact on the Efficiency of AI Search Algorithms

P.J Landin [23], in his well-known paper "*The Next 700 Programing Languages*" splits the design of programing languages into two parts: One is the choice of the written appearance of programs or their physical representation. The other is the choice of the abstract entities, such as data types, data structures, and the functional relations amongst them. These two parts are the main mechanisms of languages within a paradigm, but the main difference between paradigms is the way in which they can be used to solve problems. In another sense, in the field of programming languages, a paradigm is a style of programming that provides control over problems' solution paths. The variety of programming paradigms is a powerful tool for solving computational problems in diverse ways. The main component of programming paradigms that affects the solution of a problem is how the paradigm approaches a particular kind of problem. For example, the logic paradigm often uses recursion, unification and resolution to solve sub-goals and combine solutions.

The study of programming languages and paradigms and their performance has been a continuously changing domain with the frequent emergence of new languages (e.g. Perl, Ruby, and Python [38, 39]). Hence the evaluation of their effect on certain classes of problems such as AI search problems needs to be further examined and updated. AI search methods are often applied to alleviate the time and space complexity of NP-complete problems. As defined in the previous sections, research in AI search algorithms aims to address these two concerns by:

a) Improving heuristic evaluation functions.

b) Creating newer algorithms.

Besides these two approaches to the reduction of the time and space complexity of search problems, we would also like to be able to employ some features of programming languages or paradigms that would make an impact on the way AI search problems are solved. However, for some problem domains the way a programming paradigm solves a problem might make an impact on the time complexity of a problem. For example, for problem domains which need a lot of computation the logic paradigm is not an ideal tool. Logic programming languages are suited for symbolic computation rather than numeric computation. On the other hand the object-oriented or functional paradigm is a good choice to solve these kinds of problems. Defining and measuring the quality of code, the organization of programs, the number of lines, etc. is mostly dependent on the programmer's knowledge and her experience with the language. Hence it is quite difficult to measure these subjects. We believe quantitative evaluation of the programing paradigm is an open-ended question. However, the way diverse paradigms solve problems has had an impact on the way that software has been developed. We have a variety of programming paradigms for solving problems in diverse domains. For example, scientific applications often need only simple data structures, but need large amounts of computation. In contrast, business applications need languages that are capable of producing reports. Systems Programming languages must provide tools that provide fast execution, and allow programmers to be able to interact with hardware.

10.1. Artificial Intelligence Languages

Artificial Intelligence applications need programming paradigms that are strong for; *representation* and *search*. AI languages must provide symbolic structures to represent problem states. Symbolic computing embraces some patterns for representing states and then manipulating these symbols as opposed to only performing arithmetic calculations on states. Hence, AI programing paradigms must have tools for symbol computation, numeric computation and must also provide tools to represent problem states. Symbolic computation is crucial for AI applications. One of our Fifteen Puzzle implementations with Java shows that symbolic computation is more conveniently done with linked lists of data rather than arrays or array lists. The results show that the implementation with a linked list is faster than the implementation with an ArrayList. The built-in features of languages simplify AI applications. For example, Prolog as a Depth-First-Search tool makes its implementations shorter and more programmer independent. Within AI problem domains, some groups of problems may need more numeric computation as opposed to symbolic computation, for instance, for the implementation of problems which need to do a lot of heuristic computation. The paradigms which do not provide numeric computation or where numeric computation is a side-effect, are not a good choice for problems that need more numeric computation.

The key to writing a successful knowledge based program is the selection of an appropriate representation tool and the search mechanism(s) used. The approaches to implementation of search problems are different in different language idioms.

The four distinct programming paradigms that we are investigating:

1.) Imperative Programming

2.) Object-oriented Programming

3.) Logic Programming

4.) Multi-Paradigms

We compare the representation tools, the way the paradigms solve problems, the language idioms, and their built-in search tools. The investigation of how a programming paradigm creates nodes, represents them and how the search procedure would be defined in the paradigm, is crucial to this research. In the following sections we will give a brief description of the four programming paradigms that we have listed above.

10.2. Imperative Programming

The imperative paradigm is the oldest and it provides a model where both its programs and variables are stored together. The imperative paradigm uses a straightforward programming style. Programs contain a series of conditionals, looping statements, variables, and assignment statements. The running cycle of a program is obtained by performing a calculation which assigns values to variables, runs statements of a program until a goal is reached, or by using loops or conditionals to direct the program in a different direction. Some well-known examples of imperative languages are; C, Perl, and Cobol. Imperative languages typically have library supported data structures, such as Iterations, Vectors, Lists, Stacks, and Sets, etc.

A program in an imperative language has an fixed state. For instance, the values of variables, program counters, etc. of states are modified by constructs in the source language. As a result, such languages generally have an explicit notion of sequencing

to permit precise and deterministic control of the state changes. Imperative programs thus express how something is to be computed. The imperative paradigm is a good tool for the problem which needs sequential computation and numeric processing.

The main tools to represent problem states in the imperative languages are lists, arrays, matrices, etc. Some languages may provide more data representation tools. The main tool for giving access to problem states is iterative. We can represent a node of the Fifteen Puzzle with C as follow:

```
struct Node
{
    Int board[16];
    struct Node* parent;

};
```

After you represent the problem states, you can use loops and conditionals to create new nodes and you can also use constructs to determine the solution state. Search procedures with imperative paradigms can be done with loops and controlled with "*if* statements". By sequentially following the steps of an algorithm, new states are created and examined from presented states. The main tools to control the search are loops and conditional statements, such as if, switch, etc. The mechanism to change the state of a variable is by using the assignment statement. The assignment statement simply assigns the evaluated value to the variable. Repeating this for the all components of a state, the state of a problem changes from state *S* to *S'.*

10.3. Object-Oriented Programming (OOP) Paradigm

OOP paradigm is similar to the imperative paradigm in the sense of language idioms. It basically builds on the core structure of the imperative paradigm while adding more

components to the core of imperative paradigm. OOP paradigm provides a model in which programs are made of objects and objects perform calculations by passing messages to each other. Some well-known features of the OOP paradigm are inheritance, polymorphism, sending messages and classes. Corresponding to imperative programming languages, OOP languages also allow more library supported data structures. However there are some differences as to how the data structures are built within libraries, and the way that one creates and uses them in programs. For example, semantics and how Iterators are embedded into programs is not the same for C and Java. The time required to create objects and pass massages between them is also crucial. The technique that OOP paradigms use to solve problems is similar to the imperative paradigm in that they both manipulate sequential computations.

10.4. Logic Programming

A program in a declarative language has no implicit state. Any state information needed must be handled explicitly. A program is made up of expressions (or terms) rather than commands. Repetitive execution is accomplished by recursion rather than by sequencing. Declarative programs express what needs to be computed rather than how it is to be computed.

Logic programming is based on predicate calculus, unification, and resolution. First order predicate calculus has the representational power, and logic programming exploits this power by bringing it to the solution of AI problems. In addition, the way declarative programming languages model a problem, with clear semantics, and the

capability for high level abstraction, makes it very effective in solving problems in some AI domains.

Logic programing is good for this group of applications in:

1. Natural language processing
2. Pattern matching
3. Mathematical logic

Prolog (the name comes from **pro**gramming in **log**ic) is the language we use in this research.

The declarative paradigm is founded on two constructs, the *facts* and the database of facts, and the *rules*, or the definition of how to match certain facts from the set databases. Prolog programs look for facts from their databases that match the facts in question. Two facts match if their predicates are the same and if their corresponding arguments are the same. Prolog was not intended for numeric computation as is common for procedural or OOP languages. It is for non-numeric computation by symbolic processing. It enables the accumulation of facts and queries based on those facts. Heuristic AI search algorithms need a lot of numerical computation. Numerical computation is not a strength of logic programming. Hence, the logic paradigm is not a good choice for solving problems which needs a lot of heuristic computation. For the Fifteen Puzzle problem we need to generate nodes, compute nodes' heuristic values, and store them. The logic paradigm provides good tools for searching, generating, and tracking nodes, but it is not good for performing numeric computation. AI problem solutions involve: node generation, storing the generated nodes, and the heuristic

computation for the nodes. Hence numeric computation often comprises about 30% of the solution to problems.

11. Comparison of AI Programming Paradigms

We conclude this section with a comparison of diverse programming paradigms. We have defined a variety of search problems, and outlined the approaches to their solution in the Chapter 10. In this chapter our goal is to capture the language idioms that may give an advantage to one programming paradigm over others. Luger [25], in his book *"AI Algorithms, Data Structures, and Idioms"* has defined the language idioms as follow: "*idioms are a form and structure for knowledge that helps us a bridge the differences between patterns as an abstract descriptions of a problem and its solutions and an understanding of how best to implement that solution in a given programming language*."

Certain AI problem domains, such as constraint-satisfaction problems need programming paradigms which provide qualitative problem solving techniques rather than quantitative problem solving techniques. That is, reasoning rather than calculation. These applications need more symbolic computation rather than numeric computation. such problem domains could be amenable to solution by reduction of the initial state of the problem to some critical sub-goals. Before a goal state can be reached, first, a sub-goal must be achievable. For example the Towers of Hanoi, Eight Queens, planner problems, and Missionary-Wolf-Goal-Cabbage problems are good examples. For the solution to the Towers of Hanoi with three rings, we can employ the following strategy: To move three rings from left peg to the right peg, first you must know how to move two rings from left peg to the center peg. Before moving two rings to center peg, you must know how to move the top ring to the right peg. Logic paradigms and constraint

programming are specifically designed for these kinds of problems. Prolog is a good tool to solve this kind of problem, because prolog solves problems by dividing the main problem into sub-problem, and then it recursively solves the problems. Applying logic paradigms to these classes of problems will increase the quality of code, decrease the number of code lines, and representation of these problem with logic programming is much simpler than the other paradigms. If we make a comparison of logic programming with imperative or OOP for this group of problems, the applications of declarative paradigms would be much simpler and cleaner. However, the complexity of the solution in terms of space and time, for all the paradigms will be approximately the same. Depending to their compilers and / or libraries, some languages might generate solutions slightly faster than the other paradigms.

For search problems where the sub-goals are not clear or it is hard to define the sub-goals, the solution strategy must be different from constraint satisfaction problems. If we were to apply uninformed search algorithms to this group of problems, the solution space increases exponentially. Hence the heuristic search algorithms are the main techniques that we use to help guide the search for the purpose of reducing the solution space for this group of problems. Employing the heuristic search to generate solution to problems adds a computation factor to the solution. therefore paradigms are needed which provide symbolic and numeric computation. The impact of heuristic functions is to reduce the solution space but they need a numeric computation to do so. After the solution space becomes larger the heuristic evaluation becomes an overhead of the algorithms. The pure logic paradigm does not provide numeric computation; the numeric computation for prolog is a side-effect.

Let us compare some programming idioms of OOP and logic paradigm. A node in Fifteen Puzzle needs to record the board position (the tile position for a specific configuration), the heuristic value, the depth of node and the parent node, representation of a node of Fifteen Puzzles with Java could be as shown by figure 11.1.

```
public class Node {
   int[] board = new int[16];
   int h;
   int depth=0;
Node parent;
}
```

Figure 11.1. Represent a node For Fifteen Puzzle with Java

We can represent a node of Fifteen Puzzle with prolog as figure 11.2

```
Node(Board, H, Depth, Path).
```

Figure 11.2. Represent a node For Fifteen Puzzle with Prolog

Where, Board is the tile configuration, h is the heuristic evaluation, Depth is the depth of node, and Path the path of parent nodes. We see the representation of a node with prolog is much simpler than Java implementation.

The representation for generating a child's left node for the Fifteen Puzzle with Java can be implemented as shown in figure 11.3.

```
// Move Empty Tile to Left.
    if(!(Empty ==0 || Empty ==4 || Empty ==8 ||Empty ==12 ))       {
        newConf=new Node();
        System.arraycopy(currentConf.board, 0, newConf.board, 0, 16);
         newConf.board[Empty] = currentConf.board[Empty-1] ;
         newConf.board[Empty-1]=0;
         newConf.depth=currentConf.depth+1;
        if(!Check(newConf)){  // Check for duplication
            CheckSolutionBFS(newConf);  //Check if the node is goal state
             newConf.h=H.H(newConf.board,Goal)+newConf.depth;
            if(newConf.h<=T)
                Open.put(newConf.MoveIndex,newConf);
            else
                 min=newConf.h;    }
```

Figure 11.3. Representation of node generation with Java

The same node could be generated with Prolog as shown in figure 11.4

```
% move_left in the first row
move_left([X1,0,X3, X4,X5,X6, X7,X8,X9, X10,X11,X12,X13,X14,X15,X16], [0,X1,X3,
X4,X5,X6, X7,X8,X9, X10,X11,X12,X13,X14,X15,X16]).
move_left([X1,X2,0, X4,X5,X6, X7,X8,X9, X10,X11,X12,X13,X14,X15,X16], [X1,0,X2,
X4,X5,X6, X7,X8,X9, X10,X11,X12,X13,X14,X15,X16]).
move_left([X1,X2,X3, 0,X5,X6, X7,X8,X9, X10,X11,X12,X13,X14,X15,X16], [X1,X2,
0,X3,X5,X6, X7,X8,X9, X10,X11,X12,X13,X14,X15,X16]).
% move_left in the second row
move_left([X1,X2,X3, X4,X5,0, X7,X8,X9, X10,X11,X12,X13,X14,X15,X16], [X1,X2,X3,
X4,0,X5, X7,X8,X9, X10,X11,X12,X13,X14,X15,X16]).
move_left([X1,X2,X3, X4,X5, X6,0,X8,X9, X10,X11,X12,X13,X14,X15,X16], [X1,X2,X3,
X4,X5,X6,X8,0,X9, X10,X11,X12,X13,X14,X15,X16]).
move_left([X1,X2,X3, X4,X5, X6,X7,0,X9, X10,X11,X12,X13,X14,X15,X16], [X1,X2,X3,
X4,X5,X6,0,X7,X9,    X10,X11,X12,X13,X14,X15,X16]).
% move_left in the third row
move_left([X1,X2,X3, X4,X5, X6,X7,X8,X9, 0,X11,X12,X13,X14,X15,X16], [X1,X2,X3,
X4,X5, X6,X7,X8,0, X9,X11,X12,X13,X14,X15,X16]).
move_left([X1,X2,X3, X4,X5, X6,X7,X8,X9, X10,0,X12,X13,X14,X15,X16], [X1,X2,X3,
X4,X5, X6,X7,X8,X9, 0,X10,X12,X13,X14,X15,X16]).
move_left([X1,X2,X3, X4,X5, X6,X7,X8,X9, X10,X11,0,X13,X14,X15,X16], [X1,X2,X3,
X4,X5, X6,X7,X8,X9, X10,0,X11,X13,X14,X15,X16]).
% move_left in the fourth row
move_left([X1,X2,X3, X4,X5, X6,X7,X8,X9, X10,X11,X12,X13,0,X15,X16], [X1,X2,X3,
X4,X5, X6,X7,X8,X9, X10,X11,X12,X13,X15,0,X16]).
move_left([X1,X2,X3, X4,X5, X6,X7,X8,X9, X10,0,X12,X13,X14,0,X16], [X1,X2,X3,
X4,X5, X6,X7,X8,X9, 0,X10,X12,X13,0,X14,X16]).
move_left([X1,X2,X3, X4,X5, X6,X7,X8,X9, X10,X11,0,X13,X14,X15,0], [X1,X2,X3,
X4,X5, X6,X7,X8,X9, X10,0,X11,X13,X14,0,X15]).
```

Figure 11.4. Node generation with prolog

With prolog we can build the databases of possible moves (Left, Right, Down, and Up) see figure 11.4. The database will have 24 possible facts of moves. Whenever a node is

to be generated we must check the database of moves to see if it is possible or not. The exploration of possible nodes to be generated would add search overhead to the problem solution. For example, at depth 23 of the Fifteen Puzzle, the number of possible nodes is around 200 million. When we examine these nodes, for each node we must search for legal moves. If examining one move costs t CPU unit of time, then checking 24 possible moves would cost us $24 \times t$ seconds.

If we use the Java instead of prolog (see figure 11.3 and figure 11.4) node generation will be faster, because, the first "*if statement*" checks if it is possible to generate a move or not, thus reducing the time necessary to determine if a move is possible or not.

Conditional statements are not included in the pure logic paradigm so that we can't directly use them. However, some implementations of the logic paradigm include conditional statements as a side-effect. For example Brooklyn-Prolog has this feature. We can implement node generating with B-Prolog as shown in figure 11.5.

```
move(right,S0,S,R0,C0,Moves,MovesR):-
    C1 is C0+1,
    (C1 =< 3 ->
     update(S0,R0,C0,R0,C1,S1),
     Manhattan_dist(S1,S,0,Dist),
     Moves=[move(Dist,right,R0,C1,S1)|MovesR;
     Moves=MovesR
    ).
```

Figure 11.5. Generation of a node with B-Prolog[5] (Brooklyn Prolog) ()

[5] This code was written by Prof. Neng-Fa Zhou, March 2012 with B-Prolog

Move prevention in figure 11.5, reduces the search overheads for the pure logic paradigm, but it also loses the pure quality of the logic paradigm.

As we have described above, numerical computation is not a component of the pure logic paradigm. Path finding problem domains use heuristic search to find solutions, because heuristic searches reduce the search space. Heuristic evaluation needs to be done by numeric computation which is not included in the logic paradigm. However numeric computation is a component of the OOP and functional paradigms.

Each paradigm is specifically designed for some group of problems; for example, as we have described above, the logic paradigm is good for to solving constraint satisfaction problems. But some problem domains involve solutions that are not easily represented and solved with a single paradigm. For example, path finding problem involves numeric and symbolic computation. The solution of this group of problems can be implemented with multi-programming paradigms. For example, Fifteen Puzzle problems can be solved by employing the perimeter bidirectional search algorithm. Instead of using one paradigm we can use a mixture of paradigms. We can employ the following policy: Use a logic programming paradigm such as prolog, to initialize a BFS from start node to a defined depth *d,* and store the frontier nodes. After that, switch the search direction to the goal node and initialize a second heuristic search by employing an OOP such as Java from the goal node towards the frontiers of prolog implementation. Whenever they collide, a solution is found.

We can make the above policy more complicated by representing and tracking duplications with prolog, and generating and computing the heuristic values of nodes with Java.

Newer programing languages have tools to initialize multiple searches simultaneously. For example Java has multi-threading. With languages that support multi-threading you can initialize a regular BFS from a start node to a depth ***d***, and from that point you can use multi-threading and initialize multi-search at the same time. The experimental results show that these approaches will find a solution faster than single-threaded languages. Languages which support multi-threading perform better than other languages.

We have implemented the Donkey Puzzle with procedural C++[6], Object-Oriented C++, and Java. The results (See table 12.4.1) show that procedural C++ and OO C++ generate the same number of nodes, but the OO C++ generates results faster than procedural C++. Java also generates a solution faster than C++. JAVA and Object-Oriented C++ generated a solution in less than a second. We have used iterators and vectors for procedural C++ and we have used linked lists and iterators for Java programs.

[6] Procedural C++ employs an imperative programming style. In our implementation we have used C style of programming under the C++ platform

12. Implementations and Experimental Results

12.1. Klostki Puzzle implementation with C++ and JAVA

We implement the Klotski Puzzle with procedural C++, object oriented C++ and Java to compare the number of nodes generated, nodes expanded and the time needed to solve the problem. All the results of our experimentation are listed in table 12.4.1

We represented each node of The Klotski Puzzle with a 5-tuples as shown in figure 12.1.1.

```
struct Board
{
        int moveNumber;
        int board[5][4];
        int moveInfo[6];
        double heurMeasure;
        Board *parent;
};
```

Figure 12.1.1. C++ representation of a node of the Klotski Puzzle.

Each node included the move number and a heuristic estimate of the node's distance to the goal. Also, a parent pointer was stored with each node to trace the path to a solution. The board itself was represented with a 5 x 4 array:

```
int board[5][4] = { {120, 22, 22, 120},
                    {12, 22, 22, 12},
                    {120, 21, 21, 120},
                    {12, 11, 11, 12},
                    {11,  0,  0, 11} };
```

Figure 12.1.2. Board representation of Klotski Puzzle

Each cell stored the type of piece in that location. For example, the 2 x 2 piece was represented with four adjacent cells containing the number 22. Similarly, the 2 x 1

rectangle was represented with two adjacent cells containing the number 21, and the 1 x 1 square was represented with 11. Empty spaces were represented with 0. A 1 x 2 rectangle was represented with the number 120 in the top cell and the number 12 in the lower cell. This allowed the top half of a 1 x 2 rectangle to be distinguished from the lower half of another 1 x 2 rectangle above it. We generated new moves by finding the two empty spaces in each board state. Possible moves were generated for the pieces surrounding an empty space. If the two empty spaces were adjacent horizontally, it was determined if the 2 x 2 or the 2 x 1 piece could move up or down into the two empty spaces. Similarly, if the two empty spaces were adjacent vertically, it was determined if the 2 x 2 or a 1 x 2 piece could move right or left into the two empty spaces. The moveInfo[] array was an array of 6 numbers included with each node. The first 2 numbers store the dimensions of the piece moved; the next 2 numbers store the starting row and column of the piece, and the last 2 numbers store the new row and column of the piece after the move. For example, the array 1 2 0 0 1 0 represents a move by a 1 x 2 piece from row 0 and column 0 to row 1 and column 0. Though it may have been more efficient to store move information separately from the board structure, we included the moveInfo[ ] array with each node to avoid keeping list structures for each sequence of moves. The main lists in our C++ implementation were two list structures implemented as C++ vectors, one for newly generated nodes and one for already explored nodes. For the Java implementation we also employed two linked lists to one for newly generated nodes and the other one for nodes already expanded (Closed List.).

Each new move was compared to the two lists to check for duplication. If an identical board state was found, the less efficient path to this state was deleted, and the more efficient path was kept in the appropriate list.

	Time	Nodes Generated	Nodes Explored	Moves
BFS	209 s	24,075	24,005	116
Heur 1	86 s	17,527	14,400	122
Heur 2	11 s	6100	5077	136

Table 12.1.1. Performance of BFS and heuristic searches from start state to goal.

As shown in Table 12.1.1, we used a BFS as a base level for purposes of comparison. As expected, the BFS was slow in finding a solution, taking 209 seconds in execution time. In terms of memory usage, it generated 24,075 nodes and explored 24,005. Subsequent searches with heuristics and the bidirectional search were expected to improve upon this base level performance in both execution time and memory usage.

The BFS was also useful in finding the optimal number of moves to the goal state: 116 moves. Because the BFS is exhaustive in exploring each level of the search space before moving onto the next level, the first solution found is the optimal solution. Thus, in our representation of the puzzle, the optimal number of moves to reach a solution is 116, and this is influenced by our choice to allow pieces to move only one space at a time. When moves of more than one space are allowed, the optimal solution is closer to 80 moves. Our main heuristic was the distance between the 2 x 2's current row position and its expected row position in the goal state. The 2 x 2 is expected to reach row 3 to solve the puzzle, and we subtracted the 2 x 2's current row position from its final row position in the goal state. Thus, if the 2 x 2 was currently at row 1, the heuristic estimate

of remaining distance was calculated by simple subtraction: 3 - 1 = 2. To improve this heuristic estimate, we considered if it was possible for the 2 x 2 to move into empty space. If the 2 x 2 could move down, we subtracted 2 from the initial estimate. If it could move left or right, we subtracted 1 from the initial estimate, and if it could move up, we subtracted 0.5. If the 2 x 2 could not move into empty space, we considered the pieces immediately next to the 2 x 2. If these pieces could move, the heuristic estimate was improved by subtracting 0.5 for each movable piece. We implemented two variations of this heuristic:

Heuristic 1:

Only pieces immediately below the 2 x 2 were checked for possible moves.

Heuristic 2:

Pieces immediately below, to the left, and to the right of the 2 x 2 were checked for possible moves.

These heuristic values were multiplied by 100 to convert them to integer values. For each node, this number was added to the move number of the node, and the node with the lowest total was chosen to be explored. The search algorithm used for heuristic searches was a best-first search.

Table 12.1.1 shows the improvement in performance with Heuristic 1, finding a solution in only 86 seconds, and exploring 10,000 fewer nodes than the simple BFS. However, it required 122 moves to find a solution instead of the optimal 116. Heuristic 2 resulted in a major improvement in execution time and memory usage, finding a solution in only 11 seconds and generating ¼ as many nodes as the simple BFS. However, the number of moves was far from optimal with 136 moves rather than 116. Thus, both

heuristics led to significant improvements in execution time and memory usage, but neither produced an optimal solution in terms of number of moves.

Heuristic 2 led to much more selective generation and exploration of nodes, improving memory usage and execution time dramatically. In a trade-off typical of heuristic search, it required 136 moves to find a solution, perhaps because it explored fewer nodes and overlooked search paths which may have led to an optimal solution in 116 moves.

12.2. Performance of the Bidirectional Search

We implemented the perimeter bidirectional search by Dillenburg and Nelson [8]. The algorithm employs two steps:

1) A forward search from the start state, performed as a BFS until a frontier of nodes is reached. In our case, we varied the level of the frontier at move numbers 50, 40, or 30.

2) A backward search from the goal state, performed as a heuristic search targeting nodes in the frontier. For the search algorithm, we implemented a BFS, using two heuristics similar to Heuristic 1 and Heuristic 2 from the previous section. Because this was a backward search, we reversed the direction favored by the heuristic, and we exchanged the goal and start states. Thus, we favored moves in which the 2 x 2 moved up, and the best heuristic value was assigned to nodes with row position 0: the original position for the 2 x 2.

To assign an initial heuristic value, we simply used the current row position of the 2 x 2. If the 2 x 2 was currently at row 2, this was less favored than a node in which the 2 x 2 was at row 1 or 0. To improve this heuristic estimate, we considered if it was possible for the 2 x 2 to move into empty space. If the 2 x 2 could move up, we subtracted 2 from

the initial estimate. If it could move left or right, we subtracted 1 from the initial estimate, and if it could move down, we subtracted 0.5. If the 2 x 2 could not move into empty space, we considered the pieces immediately next to the 2 x 2. If these pieces could move, the heuristic estimate was improved by subtracting 0.5 for each movable piece. Similarly to the previous section, we implemented two variations of this heuristic:

Heuristic 1:

Only pieces immediately above the 2 x 2 were checked for possible moves.

Heuristic 2:

Pieces immediately above, to the left, and to the right of the 2 x 2 were checked for possible moves.

Using the best-first search algorithm from the previous section, we multiplied these heuristic values by 100 to convert them to integer values. For each node, this number was added to the move number of the node, and the node with the lowest total was chosen to be explored. The heuristic search reached a conclusion when it found a node in the stored frontier. At this point, the two search paths were combined by tracing backwards using the parent pointer.

	Time	**Nodes Generated**	**Nodes Explored**	**Moves**
Frontier 50	12 s	6548	6440	116
Frontier 40	1 s	3047	2945	116
Frontier 30	1 s	2438	2293	116
Frontier 20	1 s	2614	2454	116

Table 12.2.1. Performance of Bidirectional Searches with Heuristic 1.

	Time	Nodes Generated	Nodes Explored	Moves
Frontier 50	12 s	6618	6497	124
Frontier 40	1 s	3181	3033	116
Frontier 30	1 s	2590	2399	116
Frontier 20	2 s	3307	2904	116

Table 12.2.2. Performance of Bidirectional Searches with Heuristic 2.

Table 12.2.1 shows the results with Heuristic 1 for each frontier level of the bidirectional search. With the frontier at move 50, the bidirectional search performed excellently with Heuristic 1, finding a solution in the optimal 116 number of moves, and requiring only 12 seconds of execution time.

Table 12.2.2 shows the results with Heuristic 2 for each frontier level. Heuristic 2 with a frontier of 50 was less effective with 124 moves to find a solution, though its execution time and memory usage were very similar to Heuristic 1. Thus, with the frontier at move 50, the bidirectional search performed comparably to the forward-direction searches of the previous section. In particular, using Heuristic 1, the bidirectional search found a solution in 116 moves with far less memory usage and execution time than the forward-direction heuristic searches. Next, we moved the frontier back to move 40, expecting improvements in performance because of the reduced time spent in the inefficient breadth-first search. With the frontier at move 40, the bidirectional search outperformed all previous searches in all measures of performance. With both Heuristic 1 and Heuristic 2, the bidirectional search found a solution in 116 moves, requiring only a second or less of execution time. In addition, it generated and explored only 3000 nodes, halving the previous best of 6000. Heuristic 1 performed slightly better than

Heuristic 2 in terms of memory usage, generating approximately 100 fewer nodes. Testing frontiers in increments of 10, we moved the frontier back to move 30. With the frontier at move 30, performance was only slightly improved from the previous test. The main improvement was in memory usage, with about 700 fewer nodes generated and explored. Again, Heuristic 1 performed slightly better than Heuristic 2. Finally, with the frontier at move 20, there was no further improvement in performance. Rather, memory usage was less efficient: For Heuristic 1, approximately 200 more nodes were generated and explored, as compared to the previous frontier with the same heuristic.

For Heuristic 2, the memory usage was even less efficient, with approximately 800 more nodes generated and 600 more nodes explored, compared to the previous frontier with the same heuristic.

Based on these results, the bidirectional search performed better each time the frontier was moved back until reaching a frontier of 20, when memory usage deteriorated in efficiency. Comparing the effectiveness of the two heuristics, we found Heuristic 1 to be more effective than Heuristic 2 at all frontier levels. Most relevantly, the bidirectional search outperformed forward-direction search methods in execution time and memory usage, and it succeeded in finding the optimal solution in 116 moves.

12.3. Implementations in Other Programming Languages and Paradigms

Comparing implementations in procedural C++ and object-oriented C++, we found the number of nodes generated and explored to be identical for both programming styles. Thus, regardless of programming paradigm used for the implementation, the search algorithms produced the same numbers of nodes generated and explored.

Interestingly, the object-oriented C++ was much more efficient in terms of execution time, improving time performance by a factor of 2 in all searches (See Table 13.4.1). This can be explained by the different representations for the board in each implementation. In our object-oriented version, the board was represented with a node object. The fields of the node object included pointers to a board object and a move-Info object. The board object contained the 5 x 4 array of board positions, while the move-Info object contained the data associated with a particular move, such as the previous row and column of the piece making the move.

By separating the data into specific objects, we could include pointers to these objects as fields in the node class. Pointers required much less memory than the entire arrays, leading to much better time performance when traversing lists of moves to check for repeated board positions, etc. This result suggests time performance can be significantly improved by more memory efficient representations of the board, while the search algorithms are more responsible for improvement in the number of nodes generated and explored.

Different representations for the board in each implementation affect the time to generate the solution, for our Fifteen Puzzle implementation, if we switch from linked list to an arraylist, the efficiency in term of execution time was reduced.

12.4. Java vs. C++

With the Java implementation, we used Java's predefined linked list data structure for storing the lists of nodes. We use the same heuristic function 1 and algorithms as the C++ implementations. With depth 40, Java generated 2 nodes less than C++. For all

the other tests the number of nodes generated is the same. Java was much faster than procedural C++ for generating a solution. The efficiency of Java and Object oriented C++ was the same. In the initial states of the puzzle, the 2 x 2 block doesn't move much, so that Best First Search generates more nodes at beginning of the solution path. After the 2 x 2 block crosses the 2 x 1 block, with the 2 x 2 block moving more the heuristic function becomes more efficient. Hence, the Best First Search from the Start node to the Goal node generated many more nodes than a Best First Search starting from the Goal node to the Start nodes. The results show that representing data with Classes is much more efficient than representing data with pointers.

Our results also show that for problem domains which generate more nodes at the initial stage of search, employing the bidirectional search will solve problems faster, and they will use memory more efficiently. The first search will generate all frontiers by using most of the memory and leave a small portion of memory to the second search. The second search won't spend much memory and time to meet with frontiers, because most of the nodes generated by the first search. All the results are presented on the table 12.4.1.

	Time for C++	Time for Object C++	Time for Java	Nodes Generated	Nodes Explored	Moves in Solution
Breadth First Search	209 seconds	93 seconds	226 seconds	24,075 nodes	24,005 nodes	116 moves
Forward Search Heuristic 1	86 seconds	48 seconds	92 seconds	17,527 nodes	14,400 nodes	122 moves
Forward Search Heuristic 2	11 seconds	6 seconds	6 seconds	6100 nodes	5077 nodes	136 moves
Bidirectional Heuristic 1 Frontier 50	12 seconds	5 seconds	10 seconds	6548 nodes	6440 nodes	116 moves
Bidirectional Heuristic 2 Frontier 50	12 seconds	5 seconds	5 seconds	6618 nodes	6497 nodes	124 moves
Bidirectional Heuristic 1 Frontier 40	1 second	Less than 1 second	1 second	3047 nodes	2945 nodes	116 moves
Bidirectional Heuristic 2 Frontier 40	1 second	Less than 1 second	1 second	3181 nodes	3033 nodes	116 moves
Bidirectional Heuristic 1 Frontier 30	1 second	Less than 1 second	1 second	2438 nodes	2293 nodes	116 moves
Bidirectional Heuristic 2 Frontier 30	1 second	Less than 1 second	1 second	2590 nodes	2399 nodes	116 moves
Bidirectional Heuristic 1 Frontier 20	1 second	Less than 1 second	1 second	2614 nodes	2454 nodes	116 moves

Table 12.4.1. This table summarizes our studies of various search algorithms implemented with procedural C++, Object Oriented C++, and Java on the Donkey puzzle. It starts with the uninformed (row 1) exhaustive BFS and then moves to continuously more informed approaches (rows 2 and 3). Heuristic 1 checks all possible moves by the pieces below the 2 x 2 square. Heuristic 2 checks for all possible moves by pieces to the left, to the right, and then those below the 2 x 2 square. The bidirectional search

employs a BFS in the forward direction, storing all moves to a specified depth (the frontier), which ranges between 20 and 50 in increments of 10, employing the indicated heuristic in the reverse direction (rows 4 - 10). Heuristic searches starting from the goal state moving backwards to the start state are represented by rows 11 and 12. The Table presents the execution time, (columns 1 - 3) the number of nodes generated and explored (columns 4 and 5), and the number of moves needed to find a solution (column 6)[7].

12.5. Experimental Results of IBFS and BFS

When we implement Fifteen Puzzles with IBFS we represent a node with 5-tuples (see figure 12.5.1).

Each node includes the node depth, its state, and three other variables. The "MoveWay" variable is used for preventing first level duplication, the "DumFlag" variable is used for preventing deeper duplication and also used for pruning paths that are already generated. The puzzle itself was represented with a one dimensional array.

Start.board = new int [] {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

```
class Node {
   int[] board;
   int MoveWay;
   long MoveIndex;
   int DupFlag;
   int depth;
}
```

Figure 12.5.1. IBFS Node Representation of Fifteen Puzzle.

We implemented the Manhattan Distance Heuristic; the heuristic values are the number of moves a piece is away from its start or goal location. The best move is the piece

[7] We are great full to Harun Iftikahr who assisted as a part of research founded under CUNY #41 to produce this results

which is closer to its original location. We have represented a node of BFS with 4-tuples (see figure 12.5.2).

```
public class Node {
   int[] board;
   int h;
   Node parent;
   int depth;    }
```

Figure 12.5.2. BFS Node Representation of Fifteen Puzzle.

Each node includes node depth, its heuristic value, its parent node and the state of a node. The state of nodes is represented with a one-dimensional array, the same as for IBFS. The main lists in our Fifteen Puzzle implementation were linked lists. For BFS we have employed two linked lists and for IBFS we have employed one linked list.

In the first implementation of BFS each new move was compared to the Closed List. With the second implementation of BFS each new node was compared to both lists (Open and Closed) to check duplications. If an identical state was found, the less efficient path to this state was deleted and the most efficient path was kept in the appropriated list.

We employed two policies to prevent duplication nodes of IBFS;

First, by using the “MoveWay” parameter we prevented a node to generate its parent. For example if a “right” move generates a left move, this will loop back to the right movies parent node (See section 9.1.2). Whenever a repeated state has been found, we save the node which is closer to the goal or start node and prevent looping back by using the “DupFlag” variable (See section 9.1.2). The “MoveIndex” variable is employed to generate the solution path. It is the index of a goal node in the frontier list. The aim of IBFS algorithm is to find the index of a goal node into the frontier list. Once the index of

goal node found, we can generate a solution path by converting the index from decimal numbers to the base of the branching factor of the problem.

Depth	Size of Frontier List (Checks for duplicate nodes)	Time
11	3941	< 1 s
12	7820	<1 s
13	15593	1 s
14	30948	28 s
15	61173	2 m and 10 s
16	119768	9 m

Table 12.5.1. IBFS's performance and the size of frontier list by the solution depth. This version of IBFS employs a mechanism to check duplicate nodes.

Table 12.5.1 shows the results of IBFS implementation with the feature of checking duplicate nodes. If a duplicate node is found, the best node was added to the frontier list and the "DupFlag" variable is set for pruning the path to the duplicate node.

Experimental results show that, searching for duplicate nodes makes the BFS and IBFS slower than their respective versions which do not check for repeating nodes. But if IBFS or the BFS algorithm is used to generate the frontier nodes for a bidirectional search, then the second search from the goal towards the frontier nodes, spends a lot of time on heuristic calculations and searches for a possible goal node in the frontier list. Hence we should keep the frontier list size small. The size of the frontier list depends to the depth of the tree, and the time needed to generate a node, search for its duplicate

node and the time needed to save it. Hence calculation of the optimal size for the frontier list is a hard task.

Depth	Size of Frontier List (without checking duplicate nodes)	Time
11	4667	< 1 s
12	9888	<1 s
13	21047	<1
14	44973	1 s
15	95930	1 s
16	204217	2 s
17	434697	3 s
18	925983	4 s
19	1973339	10 s

Table 12.5.2. IBFS's performance and the size of frontier list by the solution depth. This version of IBFS does not employ any policy to check duplicate nodes.

Table 12.5.2 shows the results of IBFS by implementing the Fifteen Puzzle. With this implementation we can exclude the duplication check policy. As you can see from table 12.5.2 generating nodes becomes faster and the size of frontier list increases dramatically, but the problem is that there are also a lot of duplicate nodes and, if we use this version of IBFS for bidirectional search, then we have to spend much time on heuristic calculations and search as we have described above.

Depth	Size of Open List	Size of Closed	Nodes Generated	Time
11	3755	3976	7730	1 s
12	7808	7693	15501	10 s
13	15545	15501	31046	>1 m
14	30822	31045	61867	>4 m
15	60842	61866	92911	>15 m

Table 12.5.3. BFS's performance and the size of Open and Closed Lists by the solution depth. This version of BFS employs a mechanism to check duplicate nodes in Open and Closed List.

Depth	Size OF Open List	Size Of Closed	# Nodes generated	Time
11	4269	3976	8245	1 s
12	8704	8244	16948	4 s
13	17663	16948	34611	30 s
14	36177	34610	70786	>3 m
15	73006	70786	143792	>13 m

Table 12.5.4. BFS's performance and the size of Open and Closed Lists by the solution depth. This version of IBFS employs a mechanism to check duplicate nodes only in the Closed List.

Table 12.5.3 and table 12.5.4 show the results of BFS by implementing the Fifteen Puzzle. The two main differences between our IBFS and regular BFS are that: the node representation of BFS is different from our IBFS and second BFS employs a Closed List to maintain the nodes already explored and do not lead to a goal node.

The results from Tables 12.5.1, 12.5.2, 12.5.3, 12.5.4, and 12.5.5 show that the IBFS is faster for generating the frontier nodes and it uses less memory by omitting the Closed List. BFS algorithms cannot explore deeper into the search tree. This is also a drawback

of IBFS. However, IBFS can search a few steps deeper than the regular BFS. One of our experiments shows that if you omit the duplicate node check policy, IBFS can generate frontiers to depth 21. The size of node becomes larger and after a while the computer runs out of memory.

Depth	IBFS total nodes saved	BFS nodes saved
11	3941	8245
12	7820	16948
13	15593	34611
14	30948	70786
15	61173	143792
16	119768	

Table 12.5.5. Compare the space used for BFS and IBFS.

12.6. Experimental Results of IA and A**

Implementation if IA* is similar to the IBFS. The only difference is that IA* tracks heuristic evaluation of a node and the frontier list is ordered by the heuristic evaluation values. The best node is added to the front of the list.

We only present experimental results for IA* and A* algorithms which allow duplications. If we add a duplication check mechanism when we try to solve some example of the Fifteen Puzzle, the algorithms become extremely slow which is normal, because the A* algorithm usually is not employed to solve Fifteen Puzzles.

number	Goal State	Estimated Distance	Actual Distance
1	8,0,2,3,10,4,5,7,1,9,6,11,12,13,14,15	11	15
2	8,2,5,3,10,4,7,11,1,13,9,15,12,0,6,14	20	24
3	2,4,5,3,8,13,11,0,10,9,7,15,1,12,6,14	24	34
4*	13,8,14,3,9,1,0,7,15,5,4,10,12,2,6,11	29	41
5*	4,7,13,10,1,2,9,6,12,8,14,5,3,0,11,15	32	44

Table 12.6.1. Test Puzzle's Goal State configuration. The start state is {0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }. * This puzzles are taken form Korf's test puzzles.

Puzzle Number	Size Of Frontier List	Nodes Expanded	Time To Solve
1	25	44	<1s
2	59	109	1 s
3	23367	44881	1 m
4*	77532	146.772	6 m
5*	290132	570558	1 h 53 m

Table 12.6.2. Results of IA*solving puzzles' stated in table 15.6.1 s->Second, m->Minibus, h-> Hour

The IA* and A* algorithms can solve Fifteen Puzzle Problems which have a goal node not deeper than 50 moves. Table 12.6.2 summarizes the result of IA* for solving some chosen examples of the Fifteen Puzzle.

Distance	Size Of Open List	Size Of Closed List	Nodes Generated	Time To Solve
1	25	19	44	<1 s
2	59	50	109	<1 s
3	23367	21514	44981	8 s
4*	77532	69240	156772	5 m 30 s
5*				

Table 12.6.3 - Results of A* algorithm by applying it to the same group of problems from table 15.6.1. s->Second, m->Minibus, h-> Hour

Results from table 12.6.1, 12.6.2, and 12.6.3, show that IA* and A* solve the same problems in approximately the same amount of time. The only difference is that IA* uses less memory than A* by completely eliminating the Closed List.

13. Conclusions and Future Work

13.1. Conclusions

We have reviewed the main AI search algorithms, and classified them as either being unidirectional or bidirectional. Then we described various AI problem domains and the approaches that have been employed for solving them. The three main concerns of AI search problems have been described as: 1. the complexity of heuristic computation 2. Space complexity, and 3. The time required to solve them.

We have also investigated the way diverse programming paradigms are used to solve problems. We considered and compared programming paradigms by focusing on the following pertinent questions related to the general features of paradigms:

- How are different programming paradigms employed to generate solutions?
- What are the feasible and / or efficient problem representation techniques within diverse paradigms?

To this end we have implemented the Klotski Puzzle with Java, Object Oriented C++, and procedural C++. The experimental results show that there isn't a major difference between the results generated with procedural and OO languages in terms of the time required to generate a solution and the number of nodes generated.

Path finding search problems require a programming paradigm which provides tools for representation and tools for numerical computation. Most of the available programming paradigms do not provide both of these tools. For example the logic paradigm provides good tools for problem representation and also for solutions by identifying and solving

sub-problems and sub-goals. However, the logic programming paradigm does **not** have good tools for numeric computation. A mixture of paradigms would be a good tool for solving path finding problems. By employing two or more programming paradigms, one paradigm can be used for numeric computations and the other for symbolic computations. A third paradigm may even be opportunistically employed for representing the problem.

We have also introduced two newer algorithms, namely, the Indexed Breadth First Algorithm (IBFS) and the indexed A* (IA*) algorithm. These newer techniques enable the elimination of the Closed List employed by the standard BFS and A* algorithms. Our experimental results show that IBFS algorithm generates a solution faster than the standard BFS. It also needs less memory then the BFS. In addition the IA* algorithm generates a solution slightly faster than the A* algorithm and it needs less space than the A*.

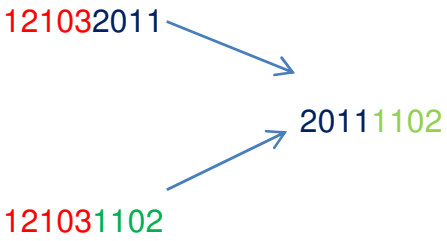
13.2. Future Work

For Fifteen Puzzle Problems heuristic evaluation functions did not produce good results, the bidirectional search algorithms generated solutions faster than the IDA* algorithm (the IDA* algorithm is the only unidirectional algorithm that can solve hard Fifteen Puzzle problems). For problems where a good heuristic estimate was found, the IDA* generated a solution faster than the bidirectional search. Unfortunately we are unable to formalize this result. However we think these results may apply to these kind (puzzles, games) of problem domains in general.

The main drawback of bidirectional search algorithms is that after two search frontiers collide, more time has to be spent finding an optimal solution. Perimeter search algorithms find a path faster than IDA*, but they are unable to find the optimal solution faster than IDA* for some group of problems. We have worked on a bidirectional search algorithm (we call it Geometrical Bidirectional Search Algorithm) to resolve this drawback of BS search algorithms. The first stage of the method works as follows: First, the perimeters are generated by an IBFS from the goal or start node to a depth *d.* If the goal node has not been found, the heuristic value for each node in the frontier list is calculated and the nodes are sorted by their heuristic values. Then the node with the minimum heuristic value is chosen as a pilot node that is called the *P* node. An IDA* search initializes from the other direction towards to the *P*. After the two searches meet at *P*, a path has been found. The length of the path is *d + y, (*see figure 14.1). The second stage of the algorithm is to find an optimal path among the perimeters. Before doing that we delete all the nodes which have a heuristic value h, such that $h \geq (d + y)$. This strategy eliminates around 18 % of the perimeter nodes. After removing these nodes, we follow the following strategy to reduce the size of the frontier list until the frontier's list size is reduced to one. The last node remaining in the frontier list is a node on the optimal path.

1. Remove the first node of IDA* list, let's call it N' (see figure 14.1).
2. Remove second node in the frontier list (N_1, see figure 14.1)
3. Find the path from pilot node *P* to N_1. This could be done in one operation by employing IBFS to create perimeters. You just have to convert the index of nodes to the base of the branching factor, subtract the same strings from the head of

the index and concatenate the remaining strings. The length of the result string is the optimal distance between two nodes. For example, If node N_i has index 121032011 and the index of node N_j is 121031102, then the distance to travel from node N_i to node N_j is;



4. Find the distance from N' to N1 (see figure 13.2)
5. If the distance from N' top N_1 < distance from P to N_1 then there is a shorter path from Goal to N' and from N' to N_1.
6. Repeat these steps until the frontier list's size is one.

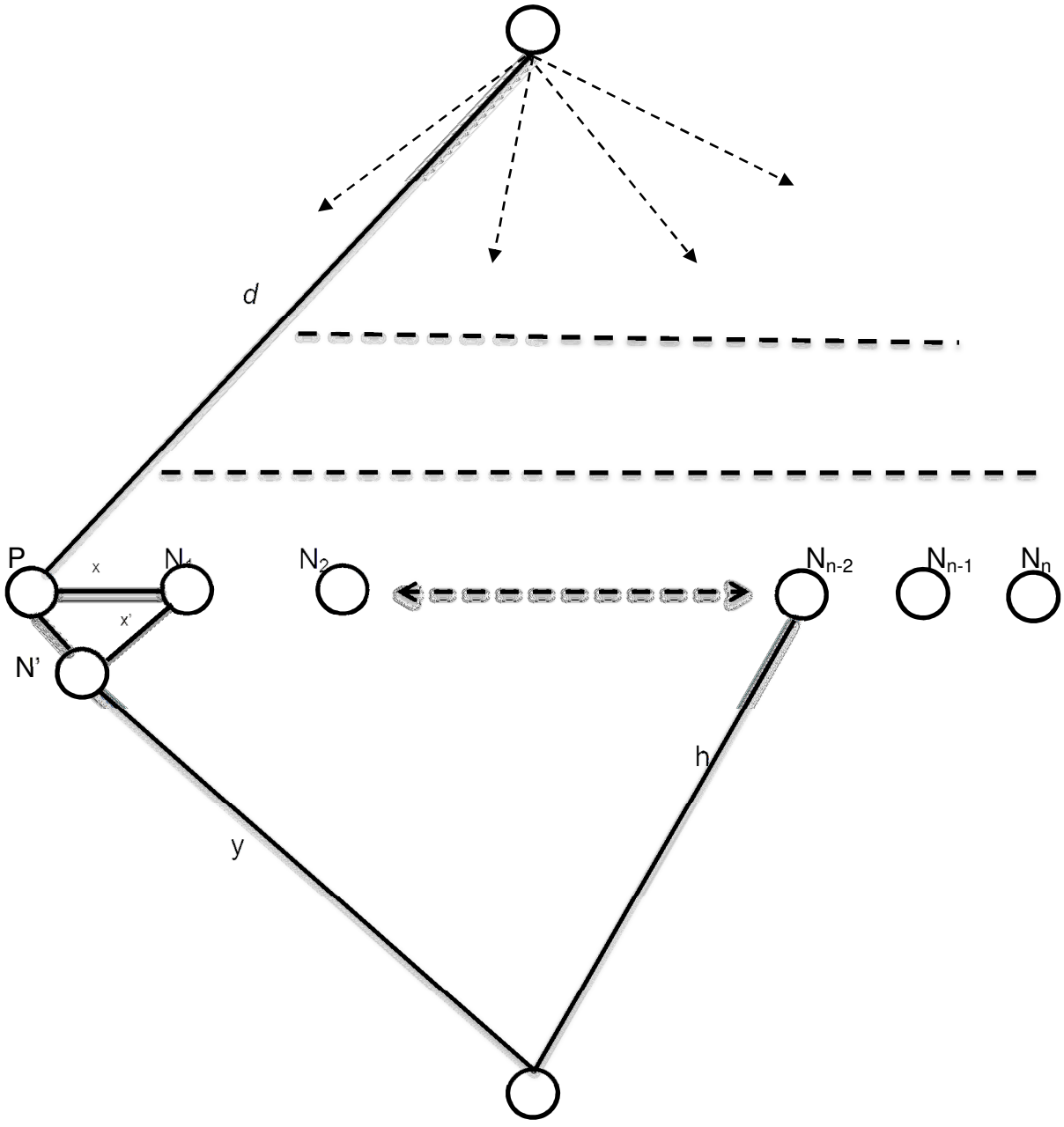


Figure 13.1. The frontier nodes generated by IBFS.

Experimental results show that implementation of this method sometimes generates the correct solution, but sometimes it fails to find the correct solution.

Hence we do not have data to support the correctness of this method. There may be some errors in the implementation or some errors in the method applied. In the future we hope to investigate this approach more deeply.

The results of this research suggest some possible directions for further studies such as implementing a path-finding problem with multi paradigms such as use of prolog to generate frontiers and use of Java for heuristic computations. The geometrical bidirectional search method that we have discussed above may have some advantage over the other BS algorithms.

14. Appendix

Appendix A.

C++ implementation of Klotski Puzzle:

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

// Structure to represent board configurations.
struct Board
{
int moveNumber;
int board[5][4];
int moveInfo[6];
double heurMeasure;
Board *parent;
};

// Global Variables.
vector<Board *> listOfMoves;       // Vector for moves generated and not explored.
vector <Board *> exploredNodes;    // Vector for moves explored in breadth-first search.
vector <Board *> exploredNodes2; // Vector for moves explored in heuristic search.
vector<Board *> move40List;        // Vector to store all nodes with moveNumber 40.
vector<Board *> solutionList;      // Vector to store the path to a solution.
bool isHeuristicSearch;
ofstream outfile("c:\\Users\\ERDALKOSE\\Desktop\\Bid4Output.txt");

// Function prototypes.
bool checkDuplicateConf(Board *&newMove);
void check1EmptySpace(Board *&currentConf, int emRow, int emCol);
void check2HorizSpaces(Board *&currentConf, int em1Row, int em1Col, int em2Row,
int em2Col);
void check2VertSpaces(Board *&currentConf, int em1Row, int em1Col, int em2Row, int
em2Col);
void printSolution(Board *&printNode, bool printRequired);
void calculateHeuristic(Board *&newConf);
void deletePath(Board *&deleteNode);
void buildSolutionList(Board *&finalNode, Board *&move40);
```

```
int main()
{
// Boards are represented with 5-row by 4-column arrays.  Each cell stores
// the dimensions of the piece in that position, such as 22 for a 2 by 2 piece.
// The 1 by 2 piece is represented with 120 to distinguish the top half of the piece
// from the lower half represented with 12.
int startLocations[5][4] = { {120, 22, 22, 120},
{12, 22, 22, 12},
{120, 21, 21, 120},
{12, 11, 11, 12},
{11, 0, 0, 11} };
// Initialize a Board structure with the starting configuration.
Board *initialConf = new Board;
initialConf->moveNumber = 0;
initialConf->heurMeasure = 0;
initialConf->parent = NULL;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
initialConf->board[i][j] = startLocations[i][j];
listOfMoves.push_back(initialConf);
int listSize = listOfMoves.size();
cout << "Initial value for listSize variable: " << listSize << endl;
Board *currentConf;
Board *tempNode;
double totalMeasure = 0;
double minHeuristic = 100;
int indexOfMin = 0;
bool printRequired;
bool stopFlag = false;
bool isFirstEmpty = true;
int em1Row, em1Col, em2Row, em2Col, firstEmIndex;
vector<Board *>::iterator iter1;
vector<Board *>::iterator iter2;
bool solutionFound;
isHeuristicSearch = false;
// Bidirectional Search Step 1
// Breadth-first search until moveNumber 40.  All nodes with moveNumber 40 are put on
move40List.
while ( !listOfMoves.empty() )
{
listSize = listOfMoves.size();
iter1 = listOfMoves.begin();

currentConf = *iter1;
exploredNodes.push_back(currentConf);
```

```
listOfMoves.erase(iter1);
if ( currentConf->moveNumber == 40 )
{
move40List.push_back(currentConf);
continue;
}
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if ( currentConf->board[i][j] == 0 )
{
if (isFirstEmpty)
{
em1Row = i;
em1Col = j;
isFirstEmpty = false;
}
// Else if this is the second empty space, the row and column are assigned to em2Row
and em2Col.
else
{
em2Row = i;
em2Col = j;
}
}
}
}
isFirstEmpty = true;
// If the empty spaces are adjacent horizontally, the function check2HorizSpaces() is
called.
if ( em1Row == em2Row && (em1Col + 1) < 4 && (em1Col + 1) == em2Col )
{
check2HorizSpaces(currentConf, em1Row, em1Col, em2Row, em2Col);
}

else if ( em1Col == em2Col && (em1Row + 1) < 5 && (em1Row + 1) == em2Row )
{
check2VertSpaces(currentConf, em1Row, em1Col, em2Row, em2Col);
}
else
{
check1EmptySpace(currentConf, em1Row, em1Col);

check1EmptySpace(currentConf, em2Row, em2Col);
}
```

```
}   // End of first while-loop and the breadth-first search.
// Next step will be a heuristic search from the solution node back to a node in
move40List.
isHeuristicSearch = true;
bool reachedMove40 = true;
int list40Size;
list40Size = move40List.size();
// Initialize a Board structure with the solution configuration.
int solutionLocations[5][4] = { {120, 120, 120, 120},
{12, 12, 12, 12},
{11, 11, 21, 21},
{0, 22, 22, 11},
{0, 22, 22, 11} };
Board *solutionConf = new Board;
solutionConf->moveNumber = 0;
solutionConf->heurMeasure = 0;
solutionConf->parent = NULL;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
solutionConf->board[i][j] = solutionLocations[i][j];
listOfMoves.push_back(solutionConf);
listSize = listOfMoves.size();
// Bidirectional Search Step 2:
// Heuristic search from the solution state to a node in move40List.
while ( !listOfMoves.empty() )
{
listSize = listOfMoves.size();
// For-loop finds the node with best heuristic measure in listOfMoves.
for (iter1 = listOfMoves.begin(); iter1 < listOfMoves.end(); iter1++)
{
tempNode = *iter1;
totalMeasure = tempNode->heurMeasure + tempNode->moveNumber;
if (totalMeasure < minHeuristic)
{
minHeuristic = totalMeasure;
iter2 = iter1;
}
}
// Node with best heuristic measure is assigned to the currentConf pointer, pushed onto
the exploredNodes2 vector,
// and popped from the listOfMoves vector.
currentConf = *iter2;
exploredNodes2.push_back(currentConf);

minHeuristic = 1000;
listOfMoves.erase(iter2);
```

```
// Test if currentConf has reached the same configuration as a node in move40List.
reachedMove40 = true;
for (iter1 = move40List.begin(); iter1 < move40List.end(); iter1++)
{
tempNode = *iter1;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
{
if ( tempNode->board[i][j] != currentConf->board[i][j] )
{
reachedMove40 = false;
}
}
if (reachedMove40)
{
solutionFound = true;
buildSolutionList(currentConf, tempNode);
iter1 = move40List.end();
}
reachedMove40 = true;
}
// If a path has been found from the solution to a node in move40List, then break out of
the while-loop.
if (solutionFound)
break;
// Otherwise, generate new moves:
// For-loops find the empty spaces in the board.
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if ( currentConf->board[i][j] == 0 )
{
// If this is the first empty space, the row and column are assigned to em1Row and
em1Col.
if (isFirstEmpty)
{
em1Row = i;
em1Col = j;
isFirstEmpty = false;
}
// Else if this is the second empty space, the row and column are assigned to em2Row
and em2Col.
else
{
em2Row = i;
```

```
em2Col = j;
}
}
}
}

isFirstEmpty = true;

// If the empty spaces are adjacent horizontally, the function check2HorizSpaces() is
called.
if ( em1Row == em2Row && (em1Col + 1) < 4 && (em1Col + 1) == em2Col )
{
check2HorizSpaces(currentConf, em1Row, em1Col, em2Row, em2Col);
}

// Else if they are adjacent vertically, check2VertSpaces() is called.
else if ( em1Col == em2Col && (em1Row + 1) < 5 && (em1Row + 1) == em2Row )
{
check2VertSpaces(currentConf, em1Row, em1Col, em2Row, em2Col);
}

// Else if the empty spaces are not adjacent, the check1EmptySpace() function is called
for each.
else
{
check1EmptySpace(currentConf, em1Row, em1Col);
check1EmptySpace(currentConf, em2Row, em2Col);
}


}   // End of second while-loop and the heuristic search from the solution to a node in
move40List.
system("pause");
return 0;
}
```

```
////////////////////////////////////////////////////////////////////////////////////
// The check2HorizSpaces() function generates possible moves when the two empty
spaces are
// adjacent horizontally.
// It has 5 parameters:
//          currentConf:      Pointer to the current board configuration.
//          em1Row:           Row of the first empty space.
//          em1Col:           Column of the first empty space.
//          em2Row:           Row of the second empty space.
//          em2Col:           Column of the second empty space.
////////////////////////////////////////////////////////////////////////////////////
void check2HorizSpaces(Board *&currentConf, int em1Row, int em1Col, int em2Row,
int em2Col)
{
Board *newConf;
bool isDuplicate;
// If the 2 by 2 piece is below the two empty spaces, a new move is generated with the 2
by 2 piece moving up one row.
if ((em1Row+1) < 4 && currentConf->board[em1Row+1][em1Col] == 22 && (em2Row+1)
< 4 && currentConf->board[em2Row+1][em2Col] == 22)
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[em1Row][em1Col] = 22;
newConf->board[em2Row][em2Col] = 22;
newConf->board[em1Row+2][em1Col] = 0;
newConf->board[em2Row+2][em2Col] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece: 2 by 2.
newConf->moveInfo[0] = 2;
newConf->moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = em1Row + 1;
newConf->moveInfo[3] = em1Col;
```

```
// moveInfo[2] and moveInfo[3] store the piece's row and column after the move.
newConf->moveInfo[4] = em1Row;
newConf->moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}

// If the 2 by 2 piece is above the two empty spaces, a new move is generated with the 2
by 2 piece moving down one row.
if ((em1Row-1) > 0 && currentConf->board[em1Row-1][em1Col] == 22 && (em2Row-1) >
0 && currentConf->board[em2Row-1][em2Col] == 22)
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j
// Board is adjusted to reflect the new move.
newConf->board[em1Row][em1Col] = 22;
newConf->board[em2Row][em2Col] = 22;
newConf->board[em1Row-2][em1Col] = 0;
newConf->board[em2Row-2][em2Col] = 0;
            // Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece: 2 by 2.
newConf->moveInfo[0] = 2;
newConf->moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = em1Row - 2;
newConf->moveInfo[3] = em1Col;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = em1Row - 1;
newConf->moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
```

```
listOfMoves.push_back(newConf);
}
}

// If the 2 by 1 piece is below the two empty spaces, a new move is generated with the 2
by 1 piece moving up one row.
if ((em1Row+1) < 5 && currentConf->board[em1Row+1][em1Col] == 21 && (em2Row+1)
< 5 && currentConf->board[em2Row+1][em2Col] == 21)
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[em1Row][em1Col] = 21;
newConf->board[em2Row][em2Col] = 21;
newConf->board[em1Row+1][em1Col] = 0;
newConf->board[em2Row+1][em2Col] = 0
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece: 2 by 1.
newConf->moveInfo[0] = 2;
newConf->moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = em1Row + 1;
newConf->moveInfo[3] = em1Col;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = em1Row;
newConf->moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
// If the 2 by 1 piece is above the two empty spaces, a new move is generated with the 2
by 1 piece moving down one row.
```

```
if ((em1Row-1)>=0 && currentConf->board[em1Row-1][em1Col] == 21 && (em2Row-
1)>=0 && currentConf->board[em2Row-1][em2Col] == 21)
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[em1Row][em1Col] = 21;
newConf->board[em2Row][em2Col] = 21;
newConf->board[em1Row-1][em1Col] = 0;
newConf->board[em2Row-1][em2Col] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece moved: 2 by 1.
newConf->moveInfo[0] = 2;
newConf->moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = em1Row - 1;
newConf->moveInfo[3] = em1Col;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = em1Row;
newConf->moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
check1EmptySpace(currentConf, em1Row, em1Col);
check1EmptySpace(currentConf, em1Row, em2Col);
}
```

```
////////////////////////////////////////////////////////////////////////////////////////
// The check2VertSpaces() function generates possible moves when the two empty spaces are
// adjacent vertically.
// It has 5 parameters:
//          currentConf:      Pointer to the current board configuration.
//          em1Row:           Row of the first empty space.
//          em1Col:           Column of the first empty space.
//          em2Row:           Row of the second empty space.
//          em2Col:           Column of the second empty space.
////////////////////////////////////////////////////////////////////////////////////////
void check2VertSpaces(Board *&currentConf, int em1Row, int em1Col, int em2Row, int em2Col)
{
Board *newConf;
bool isDuplicate;
// If the 2 by 2 piece can move left to the two empty spaces, a new move is generated with the 2 by 2 moving one column left.
if ((em1Col+1) < 3 && currentConf->board[em1Row][em1Col+1] == 22 && (em2Col+1) < 3 && currentConf->board[em2Row][em2Col+1] == 22)
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[em1Row][em1Col] = 22;
newConf->board[em2Row][em2Col] = 22;
newConf->board[em1Row][em1Col+2] = 0;
newConf->board[em2Row][em2Col+2] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece moved: 2 by 2.
newConf->moveInfo[0] = 2;
newConf->moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = em1Row;
```

```
newConf->moveInfo[3] = em1Col + 1;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = em1Row;
newConf->moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}

// If the 2 by 2 piece can move right to the two empty spaces, a new move is generated
with the 2 by 2 moving one column right.
if ((em1Col-1) > 0 && currentConf->board[em1Row][em1Col-1] == 22 && (em2Col-1) >
0 && currentConf->board[em2Row][em2Col-1] == 22)
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[em1Row][em1Col] = 22;
newConf->board[em2Row][em2Col] = 22;
newConf->board[em1Row][em1Col-2] = 0;
newConf->board[em2Row][em2Col-2] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 2 and 2 as the dimensions of the piece moved.
newConf->moveInfo[0] = 2;
newConf->moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = em1Row;
newConf->moveInfo[3] = em1Col - 2;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = em1Row;
newConf->moveInfo[5] = em1Col - 1;


if (isHeuristicSearch)
```

```
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
// If the 1 by 2 piece can move left to the two empty spaces, a new move is generated
with the 1 by 2 moving one column left.
if ((em1Col+1)<4 && currentConf->board[em1Row][em1Col+1] == 120 &&
(em2Col+1)<4 && currentConf->board[em2Row][em2Col+1] == 12)
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[em1Row][em1Col] = 120;
newConf->board[em2Row][em2Col] = 12;
newConf->board[em1Row][em1Col+1] = 0;
newConf->board[em2Row][em2Col+1] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 1 and 2 as the dimensions of the piece moved.
newConf->moveInfo[0] = 1;
newConf->moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = em1Row;
newConf->moveInfo[3] = em1Col + 1;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = em1Row;
newConf->moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
```

```
// If the 1 by 2 piece can move right to the two empty spaces, a new move is generated
with the 1 by 2 moving one column right.
if ((em1Col-1)>=0 && currentConf->board[em1Row][em1Col-1] == 120 && (em2Col-
1)>=0 && currentConf->board[em2Row][em2Col-1] == 12)
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[em1Row][em1Col] = 120;
newConf->board[em2Row][em2Col] = 12;
newConf->board[em1Row][em1Col-1] = 0;
newConf->board[em2Row][em2Col-1] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 1 and 2 as the dimensions of the piece moved.
newConf->moveInfo[0] = 1;
newConf->moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = em1Row;
newConf->moveInfo[3] = em1Col - 1;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = em1Row;
newConf->moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
// Calls to check1EmptySpace() to generate possible moves for each empty space
separately.
check1EmptySpace(currentConf, em1Row, em1Col);
check1EmptySpace(currentConf, em2Row, em2Col);
}
```

```
////////////////////////////////////////////////////////////////////////////////////
// The check1EmptySpace() function generates possible moves for an empty space.
// It has 3 parameters:
//          currentConf:      Pointer to the current board configuration.
//          emRow:            Row of the empty space.
//          emCol:            Column of the empty space.
////////////////////////////////////////////////////////////////////////////////////
void check1EmptySpace(Board *&currentConf, int emRow, int emCol)
{
Board *newConf;
int pieceRow, pieceCol, pieceType;
bool isDuplicate;
// If there is a valid space one row below the empty space.
if ( (emRow+1) < 5 )
{
// Identify piece below empty space.
pieceRow = emRow + 1;
pieceCol = emCol;
pieceType = currentConf->board[pieceRow][pieceCol];
// If the piece below the empty space is a 1 by 1 piece, generate a new move with the 1
by 1 moving up.
if ( pieceType == 11 )
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[emRow][emCol] = pieceType;
newConf->board[pieceRow][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 1 and 1 as the dimensions of the piece moved.
newConf->moveInfo[0] = 1;
newConf->moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = pieceRow;
newConf->moveInfo[3] = pieceCol;

// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
```

```
newConf->moveInfo[4] = emRow;
newConf->moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}

// If the piece below the empty space is a 1 by 2 piece, generate a new move with the 1
by 2 moving up.
if ( (emRow+2) < 5 && pieceType == 120 )
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[emRow][emCol] = pieceType;
newConf->board[pieceRow][pieceCol] = 12;
newConf->board[pieceRow+1][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 1 and 2 as the dimensions of the piece moved.
newConf->moveInfo[0] = 1;
newConf->moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = pieceRow;
newConf->moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = emRow;
newConf->moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
```

```
}

// If there is a valid space one row above the empty space.
if ( (emRow-1) >= 0 )
{
// Identify piece above empty space.
pieceRow = emRow - 1;
pieceCol = emCol;
pieceType = currentConf->board[pieceRow][pieceCol];
// If the piece above the empty space is a 1 by 1 piece, generate a new move with the 1
by 1 moving down.
if ( pieceType == 11 )
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[emRow][emCol] = pieceType;
newConf->board[pieceRow][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 1 and 1 as the dimensions of the piece moved.
newConf->moveInfo[0] = 1;
newConf->moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = pieceRow;
newConf->moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = emRow;
newConf->moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
```

```
// If the piece above the empty space is a 1 by 2 piece, generate a new move with the 1
by 2 moving down.
if ( (emRow-2) >= 0 && pieceType == 12 )
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[emRow][emCol] = pieceType;
newConf->board[pieceRow][pieceCol] = 120;
newConf->board[pieceRow-1][pieceCol] = 0;
// Call to checkDuplicateConf() board.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 1 and 2 as the dimensions of the piece moved.
newConf->moveInfo[0] = 1;
newConf->moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = pieceRow - 1;
newConf->moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = pieceRow;
newConf->moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
}
// If there is a valid space to the right of the empty space.
if ( (emCol+1) < 4 )
{
// Identify piece to the right of empty space.
pieceRow = emRow;
pieceCol = emCol + 1;
pieceType = currentConf->board[pieceRow][pieceCol];
```

```
// If the piece to the right of the empty space is a 1 by 1 piece, generate a new move
with the 1 by 1 moving left.
if ( pieceType == 11 )
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[emRow][emCol] = pieceType;
newConf->board[pieceRow][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 1 and 1 as the dimensions of the piece moved.
newConf->moveInfo[0] = 1;
newConf->moveInfo[1] = 1;

// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = pieceRow;
newConf->moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = emRow;
newConf->moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}


// If the piece to the right of the empty space is a 2 by 1 piece, generate a new move
with the 2 by 1 moving left.
if ( (emCol+2) < 4 && pieceType == 21 )
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
```

```
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[emRow][emCol] = pieceType;
newConf->board[pieceRow][pieceCol+1] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 2 and 1 as the dimensions of the moved piece.
newConf->moveInfo[0] = 2;
newConf->moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = pieceRow;
newConf->moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = emRow;
newConf->moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
}

// If there is a valid space one column to the left of the empty space.
if ( (emCol-1) >= 0 )
{
// Identify piece to the left of empty space.
pieceRow = emRow;
pieceCol = emCol - 1;
pieceType = currentConf->board[pieceRow][pieceCol];
// If the piece to the left of the empty space is a 1 by 1 piece, generate a new move with
the 1 by 1 moving right.
if ( pieceType == 11 )
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
```

```
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[emRow][emCol] = pieceType;
newConf->board[pieceRow][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 1 and 1 as the dimensions of the piece moved.
newConf->moveInfo[0] = 1;
newConf->moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = pieceRow;
newConf->moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = emRow;
newConf->moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}


// If the piece to the left of the empty space is a 2 by 1 piece, generate a new move with
the 2 by 1 moving right.
if ( (emCol-2) >= 0 && pieceType == 21 )
{
newConf = new Board;
newConf->moveNumber = currentConf->moveNumber + 1;
newConf->parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf->board[i][j] = currentConf->board[i][j];
// Board is adjusted to reflect the new move.
newConf->board[emRow][emCol] = pieceType;
newConf->board[pieceRow][pieceCol-1] = 0;
```

```
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (isDuplicate)
delete newConf;
else
{
// moveInfo[0] and moveInfo[1] store 2 and 1 as the dimensions of the piece moved.
newConf->moveInfo[0] = 2;
newConf->moveInfo[1] = 1;

// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf->moveInfo[2] = pieceRow;
newConf->moveInfo[3] = pieceCol - 1;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf->moveInfo[4] = pieceRow;
newConf->moveInfo[5] = pieceCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf->heurMeasure = 0;
listOfMoves.push_back(newConf);
}
}
}
}



////////////////////////////////////////////////////////////////////////////////
// The checkDuplicateConf() function returns true if newMove is a duplicate of any
// previous configuration, and if newMove has taken more moves to reach this
configuration.
//
// If newMove is a duplicate with fewer moves, then newMove is added to listOfMoves,
// and the deletePath() function is called to delete the less efficient path.
// It has one parameter:
//        newMove:      A pointer to a new configuration.
////////////////////////////////////////////////////////////////////////////////
bool checkDuplicateConf(Board *&newMove)
{
bool flag = true;
vector<Board *>::iterator iter;
Board *tempNode;

// Check if newMove is already contained in listOfMoves.
for (iter = listOfMoves.begin(); iter < listOfMoves.end(); iter++)
```

```
{
tempNode = *iter;
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if (newMove->board[i][j] != tempNode->board[i][j])
flag = false;
}
}
if (flag == true)
{
// If newMove reached this configuration in fewer moves, call deletePath() on the
duplicate in listOfMoves.
if ( newMove->moveNumber < tempNode->moveNumber )
{
listOfMoves.erase(iter);
deletePath(tempNode);
return false;
}
else
{
return true;
}
}

flag = true;
}


// If this is the breadth-first search, then check the exploredNodes vector.
if (!isHeuristicSearch)
{
// Check if newMove is already contained in exploredNodes.
for (iter = exploredNodes.begin(); iter < exploredNodes.end(); iter++)
{
tempNode = *iter;

for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if (newMove->board[i][j] != tempNode->board[i][j])
flag = false;
}
}
```

```
if (flag == true)
{
// If newMove reached this configuration in fewer moves, call deletePath() on the
duplicate in exploredNodes.
if (newMove->moveNumber < tempNode->moveNumber)
{
exploredNodes.erase(iter);
deletePath(tempNode);
return false;
}
else
{
return true;
}
}
flag = true;
}
}


// Else if this is the heuristic search, check the exploredNodes2 vector.
else
{
// Check if newMove is already contained in exploredNodes2.
for (iter = exploredNodes2.begin(); iter < exploredNodes2.end(); iter++)
{
tempNode = *iter;

for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if (newMove->board[i][j] != tempNode->board[i][j])
flag = false;
}
}

if (flag == true)
{
// If newMove reached this configuration in fewer moves, call deletePath() on the
duplicate in exploredNodes2.
if (newMove->moveNumber < tempNode->moveNumber)
{
exploredNodes2.erase(iter);
deletePath(tempNode);
```

```
return false;
}
else
{
return true;
}
}

flag = true;
}
}


return false;

}




////////////////////////////////////////////////////////////////////////////////////
// The deletePath() function deletes less efficient paths when a duplicate configuration
// is found by the checkDuplicateConf() function.
// It has one parameter:
//        deleteNode:   Pointer to the node to be deleted.
////////////////////////////////////////////////////////////////////////////////////
void deletePath(Board *&deleteNode)
{
Board *tempNode;

vector<Board *>::iterator iter;

// Check listOfMoves for nodes which are descendants of deleteNode.
if ( !listOfMoves.empty() )
{
for (iter = listOfMoves.begin(); iter < listOfMoves.end(); iter++)
{
tempNode = *iter;

if ( tempNode->parent != NULL && tempNode->parent == deleteNode )
{
listOfMoves.erase(iter);
deletePath(tempNode);
}
}
}
```

```
// If this is the breadth-first search, check exploredNodes for nodes which are
descendants of deleteNode.
if ( !isHeuristicSearch && !exploredNodes.empty() )
{
for (iter = exploredNodes.begin(); iter < exploredNodes.end(); iter++)
{
tempNode = *iter;

if ( tempNode->parent != NULL && tempNode->parent == deleteNode )
{
exploredNodes.erase(iter);
deletePath(tempNode);
} } }


// If this is the heuristic search, check exploredNodes2 for nodes which are descendants
of deleteNode.
if ( isHeuristicSearch && !exploredNodes2.empty() )
{
for (iter = exploredNodes2.begin(); iter < exploredNodes2.end(); iter++)
{
tempNode = *iter;

if ( tempNode->parent != NULL && tempNode->parent == deleteNode )
{
exploredNodes2.erase(iter);
deletePath(tempNode);
}  } }


delete deleteNode;
deleteNode = NULL;
}
```

```
////////////////////////////////////////////////////////////////////////////////
// The printSolution() function prints information about a configuration and the sequence
// of moves leading to this configuration.
// It has two parameters:
//        printNode:     A pointer to the configuration to be printed.
//        printRequired: A boolean flag set to true when printing the entire sequence of
//                       moves, and set to false when printing only the configuration.
////////////////////////////////////////////////////////////////////////////////
void printSolution(Board *&printNode, bool printRequired)
{
Board *tempNode;
int solutionSize;
// Print out the configuration.
outfile << "Final configuration:\n";
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
{
if ( printNode->board[i][j] == 11 )
outfile << "The 1 by 1 piece is located at row " << i << " and column " << j << ".\n";
else if ( printNode->board[i][j] == 21 && (j+1) < 4 && printNode->board[i][j+1] == 21 )
outfile << "The 2 by 1 piece is located at row " << i << " and column " << j << ".\n";
else if ( printNode->board[i][j] == 120 )
outfile << "The 1 by 2 piece is located at row " << i << " and column " << j << ".\n";
else if ( printNode->board[i][j] == 22 && (i+1) < 5 && printNode->board[i+1][j] == 22
                                                    && (j+1) < 4 && printNode->board[i][j+1] == 22 )
outfile << "The 2 by 2 piece is located at row " << i << " and column " << j << ".\n";
}
outfile << "\n\n";
{
solutionSize = solutionList.size();
outfile << "At the end of the program, the size of the solutionList is " << solutionSize <<
"\n\n";
outfile << "\n\nThe sequence of moves leading to this configuration is:\n\n";
// For-loop prints out the sequence of moves.
for (int i = 1; i < solutionSize - 1; i++)
{
outfile << "Move " << solutionList[i]->moveNumber << ": The ";

outfile << solutionList[i]->moveInfo[0] << " by " << solutionList[i]->moveInfo[1];
outfile << " piece was moved from row ";
// If the moveNumber is 40 or less, moveInfo[2] and moveInfo[3] are the starting row and
column for the piece,
// and moveInfo[4] and moveInfo[5] are the new row and column after the move.
if (solutionList[i]->moveNumber <= 40)
{
outfile << solutionList[i]->moveInfo[2];
```

```
outfile << " and column " << solutionList[i]->moveInfo[3];
outfile << " to row " << solutionList[i]->moveInfo[4];
outfile << " and column " << solutionList[i]->moveInfo[5];
outfile << endl << endl;
}

/ Else if moveNumber is more than 40, moveInfo[4] and moveInfo[5] are the starting row
and column for the piece,
else
{
outfile << solutionList[i]->moveInfo[4];
outfile << " and column " << solutionList[i]->moveInfo[5];
outfile << " to row " << solutionList[i]->moveInfo[2];
outfile << " and column " << solutionList[i]->moveInfo[3];
outfile << endl << endl;
}
}
outfile << "\n\n";
solutionList.clear();
}
}

////////////////////////////////////////////////////////////////////////////////
// The calculateHeuristic() function assigns a heuristic estimate of the new move's
distance
// to the solution.
// It has one parameter:
//        newConf:      Pointer to the new board.
////////////////////////////////////////////////////////////////////////////////
void calculateHeuristic(Board *&newConf)
{
int squareRow, squareCol;
int em1Row, em1Col, em2Row, em2Col, firstEmIndex;
int pieceRow, pieceCol;
bool isFirstEmpty = true;
Board *tempNode;


// For-loops find the row and column of the 2 by 2 square in this configuration.
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
{
if (newConf->board[i][j] == 22 && (i+1) < 5 && newConf->board[i+1][j] == 22 && (j+1) < 4
&& newConf->board[i][j+1] == 22)
{
squareRow = i;
```

```
squareCol = j;
break;
}
}



// Heuristic is assigned by taking the 2 by 2 square's current row: Row 0 is ideal to reach
the starting position.
newConf->heurMeasure = squareRow;
// For-loops find the empty spaces in the new board.
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
{
if ( newConf->board[i][j] == 0 )
{
// If this is the first empty space, the row and column are assigned to em1Row and
em1Col.
if (isFirstEmpty)
{
em1Row = i;
em1Col = j;
isFirstEmpty = false;
}

// Else if this is the second empty space, the row and column are assigned to em2Row
and em2Col.
else
{
em2Row = i;
em2Col = j;
}
}
}

      isFirstEmpty = true;

// If the 2 by 2 square can move up one row, improve its heuristic score by subtracting 2.
if ( (squareRow-1) >= 0 && em1Row == (squareRow-1) && em2Row == (squareRow-1)
&& em1Col == squareCol && (squareCol+1) < 4 && em2Col == (squareCol+1) )
newConf->heurMeasure -= 2.0;
// Else if it can move down one row, improve its heuristic score by subtracting 0.5.
else if ( (squareRow+2) < 5 && em1Row == (squareRow+2) && em2Row ==
(squareRow+2) && em1Col == squareCol
                                          && (squareCol+1) < 4 && em2Col ==
                                    (squareCol+1) )
```

```
newConf->heurMeasure -= 0.5;

// Else if it can move right one column, improve its heuristic score by subtracting 1.
else if ( (squareRow+1) < 5 && em1Row == squareRow && em2Row == (squareRow+1)
&& (squareCol+2) < 4 && em1Col == (squareCol+2) && em2Col == (squareCol+2) )
newConf->heurMeasure -= 1;
// Else if it can move left one column, improve its heuristic score by subtracting 1.
else if ( (squareRow+1) < 5 && em1Row == squareRow && em2Row == (squareRow+1)
&& (squareCol-1) >= 0 && em1Col == (squareCol-1) && em2Col == (squareCol-1) )
newConf->heurMeasure -= 1;

// Else if there are pieces above the 2 by 2, check possible moves for these pieces.
else if ( (squareRow-1) >= 0 )
{
pieceRow = squareRow - 1;

for (int i = 0; i < 2; i++)
{
if (i == 0)
pieceCol = squareCol;
else if (i == 1)
pieceCol = squareCol + 1;

// If the piece above the 2 by 2 is a 1 by 1 piece, check possible moves for the 1 by 1.
if ( pieceCol < 4 && newConf->board[pieceRow][pieceCol] == 11 )
{

// If the 1 by 1 can move right or left, the heuristic estimate is improved by subtracting
0.5.
if ( em1Row == pieceRow && (pieceCol+1) < 4 && em1Col == (pieceCol+1) )
newConf->heurMeasure -= 0.5;
else if ( em1Row == pieceRow && (pieceCol-1) >= 0 && em1Col == (pieceCol-1) )
newConf->heurMeasure -= 0.5;
else if ( em2Row == pieceRow && (pieceCol+1) < 4 && em2Col == (pieceCol+1) )
newConf->heurMeasure -= 0.5;
else if ( em2Row == pieceRow && (pieceCol-1) >= 0 && em2Col == (pieceCol-1) )
newConf->heurMeasure -= 0.5;
// If the 1 by 1 can move up, the heuristic estimate is improved by subtracting 0.5.
if ( (pieceRow-1) >= 0 && em1Col == pieceCol && em1Row == (pieceRow-1) )
newConf->heurMeasure -= 0.5;
else if ( em2Col == pieceCol && (pieceRow-1) >= 0 && em2Row == (pieceRow-1) )
newConf->heurMeasure -= 0.5;
}
// If the piece above the 2 by 2 is a 1 by 2 piece, check possible moves for the 1 by 2.
else if ( pieceCol < 4 && newConf->board[pieceRow][pieceCol] == 120 )
{
```

```
// If the 1 by 2 can move up, the heuristic estimate is improved by subtracting 0.5.
if ( em1Col == pieceCol && (pieceRow-1) >= 0 && em1Row == (pieceRow-1) )
newConf->heurMeasure -= 0.5;
else if ( em2Col == pieceCol && (pieceRow-1) >= 0 && em2Row == (pieceRow-1) )
newConf->heurMeasure -= 0.5;
// If the 1 by 2 can move left or right, the heuristic estimate is improved by subtracting
0.5.
if ( em1Row == pieceRow && (pieceRow+1) < 5 && em2Row == (pieceRow+1) &&
(pieceCol+1) < 4 && em1Col == (pieceCol+1) && em2Col == (pieceCol+1) )
newConf->heurMeasure -= 0.5;
else if ( em1Row == pieceRow && (pieceRow+1) < 5 && em2Row == (pieceRow+1) &&
(pieceCol-1) >= 0 && em1Col == (pieceCol-1) && em2Col == (pieceCol-1) )
newConf->heurMeasure -= 0.5;
}
// If the piece above the 2 by 2 is a 2 by 1 piece, check possible moves for the 2 by 1.
else if ( pieceCol < 4 && newConf->board[pieceRow][pieceCol] == 21 )
{
// If the 2 by 1 can move up, the heuristic estimate is improved by subtracting 1.
if ( (pieceRow-1) >= 0 && em1Row == (pieceRow-1) && em2Row == (pieceRow-1) &&
em1Col == pieceCol && (pieceCol+1) < 4 && em2Col == (pieceCol+1) )
newConf->heurMeasure -= 1;
else
{
// If the 2 by 1 can move left or right, the heuristic estimate is improved by subtracting
0.5.
if ( em1Row == pieceRow && (pieceCol+2) < 4 && em1Col == (pieceCol+2) )
newConf->heurMeasure -= 0.5;
else if ( em1Row == pieceRow && (pieceCol-1) >= 0 && em1Col == (pieceCol-1) )
newConf->heurMeasure -= 0.5;
if ( em2Row == pieceRow && (pieceCol+2) < 4 && em2Col == (pieceCol+2) )
newConf->heurMeasure -= 0.5;
else if ( em2Row == pieceRow && (pieceCol-1) >= 0 && em2Col == (pieceCol-1) )
newConf->heurMeasure -= 0.5;
}
}
}       // For-loop's closing brace
}
// Multiply the heurMeasure by 100 to make it an integer value.
newConf->heurMeasure *= 100.0;
}
```

```
////////////////////////////////////////////////////////////////////////////////
// The buildSolutionList() function combines the two paths found by the bidirectional
search
// into a single path from the start state to a solution.
// It has two parameters:
//        finalNode:    Pointer to the last node explored by the heuristic search.
//        move40:       Pointer to the node in move40List which is identical to finalNode.
////////////////////////////////////////////////////////////////////////////////
void buildSolutionList(Board *&finalNode, Board *&move40)
{
Board *tempNode;

int solutionSize;

int count;

vector<Board *> tempList;

// Build the path from move40 back to the start state.
tempList.push_back(move40);
tempNode = move40->parent;
while (tempNode != NULL)
{
tempList.push_back(tempNode);
tempNode = tempNode->parent;
}

// Transfer this path from tempList to solutionList.
for (int i = tempList.size() - 1; i >= 0; i--)
{
solutionList.push_back(tempList[i]);
}


// Assign 41 as the moveNumber for finalNode, and build the path from finalNode to the
solution state.
finalNode->moveNumber = 41;
solutionList.push_back(finalNode);
tempNode = finalNode->parent;
count = 2;
while (tempNode != NULL)
{
tempNode->moveNumber = 40 + count;
solutionList.push_back(tempNode);
```

```
tempNode = tempNode->parent;
count++;
}
cout << "The size of the solutionList is " << solutionList.size() << ".\n\n";
solutionSize = solutionList.size();
// Call to printSolution() to print out the solution path.
tempNode = solutionList[solutionSize - 1];
printSolution(tempNode, true);
}
```

Appendix B

Java implementation of Klotski Puzzle

```
import java.util.*;
//Board Representation of Klotski Puzzlepublic class Board
 {
       int moveNumber;
       int[][] board = new int[5][4];
      int [] moveInfo = new int[6];
      double heurMeasure;
      Board parent;
}
import java.util.*;
import java.io.*;
import java.lang.*;
 public class Search
{
    // Boards are represented with 5-row by 4-column arrays.  Each cell stores
    // the dimensions of the piece in that position, such as 22 for a 2 by 2 piece.
    // The 1 by 2 piece is represented with 120 to distinguish the top half of the piece
  // from the lower half represented with 12.
//lists
List <Board> listOfMoves  = new LinkedList();
List <Board> exploredNodes  = new LinkedList();
List <Board> exploredNodes2  = new LinkedList();
List <Board> move40List  = new LinkedList();
List  <Board> solutionList  = new LinkedList();
 boolean isHeuristicSearch;
 int SizeExp1=0;
 int SizeExp2=0;
 int SizeMl=0;
 FileWriter fstream;
 BufferedWriter out;
//
public  void  BidirectionalSearch() {
int [][] startLocations = new int [][]
{{120,22,22,120},{12,22,22,12},{120,21,21,120},{12,11,11,12},{11,0,0,11} };
// Initialize a Board structure with the starting configuration.
Board initialConf = new Board();
initialConf.moveNumber = 0;
initialConf.heurMeasure = 0;
initialConf.parent = null;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
```

```
initialConf.board[i][j] = startLocations[i][j];
listOfMoves.add(initialConf);
int listSize = listOfMoves.size();
System.out.println("Initial value for listSize variable: "+ listSize);

import java.util.*;
import java.io.*;
import java.lang.*;
public class Search
{
//
// Boards are represented with 5-row by 4-column arrays.  Each cell stores
// the dimensions of the piece in that position, such as 22 for a 2 by 2 piece.
// The 1 by 2 piece is represented with 120 to distinguish the top half of the piece
// from the lower half represented with 12.
//lists
List <Board> listOfMoves  = new LinkedList();
List <Board> exploredNodes  = new LinkedList();
List <Board> exploredNodes2  = new LinkedList();
List <Board> move40List  = new LinkedList();
List  <Board> solutionList  = new LinkedList();
boolean isHeuristicSearch;
int SizeExp1=0;
int SizeExp2=0;
int SizeMl=0;
FileWriter fstream;
BufferedWriter out;
//
public  void  BidirectionalSearch() {
int [][] startLocations = new int [][]
{{120,22,22,120},{12,22,22,12},{120,21,21,120},{12,11,11,12},{11,0,0,11} };
// Initialize a Board structure with the starting configuration.
Board initialConf = new Board();
initialConf.moveNumber = 0;
initialConf.heurMeasure = 0;
initialConf.parent = null;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
initialConf.board[i][j] = startLocations[i][j];
  listOfMoves.add(initialConf);
   int listSize = listOfMoves.size();
   System.out.println("Initial value for listSize variable: "+ listSize);
    Board currentConf=new Board();
    Board tempNode=new Board();
    double totalMeasure = 0;
    double minHeuristic = 100;
```

```
    int indexOfMin = 0;
    boolean printRequired;
    boolean stopFlag = false;
    boolean isFirstEmpty = true;
    int em1Row=0;
    int em1Col=0;
    int em2Row=0;
    int   em2Col=0;
    int firstEmIndex=0;
    int p=0;
    boolean solutionFound=false;
   isHeuristicSearch = false;

// Bidirectional Search Step 1:
// Breadth-first search until moveNumber 40.  All nodes with moveNumber 40 are put on
move40List.
//for (Board Lst: listOfMoves)
long start = System.currentTimeMillis();
System.out.println("Start time is :"+start+"\n");
while ( !listOfMoves.isEmpty())
{
currentConf=listOfMoves.get(0);
listSize = listOfMoves.size();
// Assign the first node in listOfMoves to the currentConf.
exploredNodes.add(currentConf);
listOfMoves.remove(currentConf);
// If the moveNumber is 40, the node is placed on move40List without generating further
moves.
if ( currentConf.moveNumber ==40 )
{
move40List.add(currentConf);
continue;
}

// Otherwise, generate new moves:
// For-loops find the empty spaces in the board.
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if ( currentConf.board[i][j] == 0 )

// If this is the first empty space, the row and column are assigned to em1Row and
em1Col.
if (isFirstEmpty)
{
```

```
em1Row = i;
em1Col = j;
isFirstEmpty = false;
}
// Else if this is the second empty space, the row and column are assigned to em2Row
and em2Col.
else
{
em2Row = i;
em2Col = j;
}
}
}
}
isFirstEmpty = true;
// If the empty spaces are adjacent horizontally, the function check2HorizSpaces() is
called.
if ( em1Row == em2Row && (em1Col + 1) < 4 && (em1Col + 1) == em2Col )
{
check2HorizSpaces(currentConf, em1Row, em1Col, em2Row, em2Col);
}

// Else if they are adjacent vertically, check2VertSpaces() is called.
else if ( em1Col == em2Col && (em1Row + 1) < 5 && (em1Row + 1) == em2Row )
{
check2VertSpaces(currentConf, em1Row, em1Col, em2Row, em2Col);
}
// Else if the empty spaces are not adjacent, the check1EmptySpace() function is called
for each.
else
{

check1EmptySpace(currentConf, em1Row, em1Col);
check1EmptySpace(currentConf, em2Row, em2Col);
}


}   // End of first while-loop and the breadth-first search.
for(Board lst: move40List){
try {
fstream = new FileWriter("output.txt",true);
out= new BufferedWriter(fstream);
out.write("heuristic messure: "+lst.moveNumber);
out.newLine();
for (int m=0; m<6;m++){
out.write("corrent node move info : "+lst.moveInfo[m]);
```

```
out.newLine();
}
out.write("corrent node move number: "+lst.moveNumber);
out.newLine();
out.write("elements of move 40 list "+ move40List.size());
out.newLine();
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++){
out.write(lst.board[i][j]+" ");
if(j==3)
out.newLine();
}

out.write("----------------------");
out.newLine();
out.close();
}catch(IOException e){       System.out.println("There was a problem:" + e);
}
}
TempNode1=move40List.get(0);
buildSolutionList(TempNode1);
// Next step will be a heuristic search from the solution node back to a node in
move40List.
isHeuristicSearch = true;
Board TempNode1= new Board();
boolean reachedMove40 = true;
int list40Size;
list40Size = move40List.size();
// Initialize a Board structure with the solution configuration.
int [][] solutionLocations = new int [][]
{{120,120,120,120},{12,12,12,12},{11,11,21,21},{0,22,22,11},{0,22,22,11} };
Board solutionConf = new Board();
solutionConf.moveNumber = 0;
solutionConf.heurMeasure = 0;
solutionConf.parent = null;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
solutionConf.board[i][j] = solutionLocations[i][j];
listOfMoves.add(solutionConf);
listSize = listOfMoves.size();
// Bidirectional Search Step 2:
// Heuristic search from the solution state to a node in move40List.
while ( !listOfMoves.isEmpty() )
{
listSize = listOfMoves.size();
```

```
// For-loop finds the node with best heuristic measure in listOfMoves.
for (Board Lst: listOfMoves)  {
totalMeasure = Lst.heurMeasure + Lst.moveNumber;
if (totalMeasure < minHeuristic)
{
minHeuristic = totalMeasure;
currentConf = Lst;
}  }
// Node with best heuristic measure is assigned to the currentConf, pushed onto the
exploredNodes2 vector,
// and popped from the listOfMoves vector.
//currentConf = TempNode1;
exploredNodes2.add(currentConf);
minHeuristic = 1000;
listOfMoves.remove(currentConf);
// Test if currentConf has reached the same configuration as a node in move40List.
reachedMove40 = true;
for (Board Lst1: move40List)
{
tempNode = Lst1;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
{
if ( Lst1.board[i][j] != currentConf.board[i][j] )
{
reachedMove40 = false;
} }

if (reachedMove40)
{
solutionFound = true;
buildSolutionList(currentConf, Lst1);
tempNode = move40List.get(move40List.size()-1);
}
reachedMove40 = true;
}
// If a path has been found from the solution to a node in move40List, then break out of
the while-loop.
if (solutionFound)
{
System.out.println("the size of Explored Node List is  : "+exploredNodes.size()+"\n");
System.out.println("the size of  Explored 2 Node list is :"+exploredNodes2.size()+"\n");
System.out.println(" the size of List Of Moves is:
"+(listOfMoves.size()+exploredNodes.size()+exploredNodes2.size()+SizeExp1+SizeExp
2+SizeMl)+"\n");
```

```
System.out.println(" the size of List OfExplored is:
"+(exploredNodes.size()+exploredNodes2.size()+SizeExp1+SizeExp2)+"\n");
System.out.println("dublication for list of Explored nodes: "+SizeExp1+"\n" );
System.out.println("Dublication for list of Explored 2: "+ SizeExp2+"\n");
System.out.println(" Dublication for List Of Moves: "+SizeMl+"\n");
long time = System.currentTimeMillis() - start;
System.out.println(" Time to run is :"+ time/1000+" seconds. \n");
break;
}
// Otherwise, generate new moves:
// For-loops find the empty spaces in the board.
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if ( currentConf.board[i][j] == 0 )
{
// If this is the first empty space, the row and column are assigned to em1Row and
em1Col.
if (isFirstEmpty)
{
em1Row = i;
em1Col = j;
isFirstEmpty = false;
}
// Else if this is the second empty space, the row and column are assigned to em2Row
and em2Col.
else
{
em2Row = i;
em2Col = j;
} } } }
isFirstEmpty = true;
 // If the empty spaces are adjacent horizontally, the function check2HorizSpaces() is
called.
if ( em1Row == em2Row && (em1Col + 1) < 4 && (em1Col + 1) == em2Col )
{
check2HorizSpaces(currentConf, em1Row, em1Col, em2Row, em2Col);
}
// Else if they are adjacent vertically, check2VertSpaces() is called.
else if ( em1Col == em2Col && (em1Row + 1) < 5 && (em1Row + 1) == em2Row )
{
check2VertSpaces(currentConf, em1Row, em1Col, em2Row, em2Col);
}
// Else if the empty spaces are not adjacent, the check1EmptySpace() function is called
for each.
```

```
else
{
check1EmptySpace(currentConf, em1Row, em1Col);
check1EmptySpace(currentConf, em2Row, em2Col);
}
}   // End of second while-loop and the heuristic search from the solution to a node in
move40List.
}

////////////////////////////////////////////////////////////////////////////////////////
// The check2HorizSpaces() function generates possible moves when the two empty
spaces are
// adjacent horizontally.
// It has 5 parameters:
//          currentConf:      Current board configuration.
//          em1Row:           Row of the first empty space.
//          em1Col:           Column of the first empty space.
//          em2Row:           Row of the second empty space.
//          em2Col:           Column of the second empty space.
////////////////////////////////////////////////////////////////////////////////////////
public void check2HorizSpaces(Board currentConf, int em1Row, int em1Col, int
em2Row, int em2Col)
{
Board newConf;
 boolean isDuplicate;
 // If the 2 by 2 piece is below the two empty spaces, a new move is generated with the
2 by 2 piece moving up one row.
if ((em1Row+1) < 4 && currentConf.board[em1Row+1][em1Col] == 22 && (em2Row+1)
< 4 && currentConf.board[em2Row+1][em2Col] == 22)
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;

newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
 // Board is adjusted to reflect the new move.
newConf.board[em1Row][em1Col] = 22;
 newConf.board[em2Row][em2Col] = 22;

newConf.board[em1Row+2][em1Col] = 0;
 newConf.board[em2Row+2][em2Col] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
 if (!isDuplicate)
```

```
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece: 2 by 2.
newConf.moveInfo[0] = 2;
 newConf.moveInfo[1] = 2;
 // moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = em1Row + 1;
 newConf.moveInfo[3] = em1Col;
// moveInfo[2] and moveInfo[3] store the piece's row and column after the move.
newConf.moveInfo[4] = em1Row;
newConf.moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
}
}

// If the 2 by 2 piece is above the two empty spaces, a new move is generated with the 2
by 2 piece moving down one row.
if ((em1Row-1) > 0 && currentConf.board[em1Row-1][em1Col] == 22 && (em2Row-1) >
0 && currentConf.board[em2Row-1][em2Col] == 22)
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[em1Row][em1Col] = 22;
newConf.board[em2Row][em2Col] = 22;
newConf.board[em1Row-2][em1Col] = 0;
newConf.board[em2Row-2][em2Col] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece: 2 by 2.
newConf.moveInfo[0] = 2;

newConf.moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = em1Row - 2;
newConf.moveInfo[3] = em1Col;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
```

```
newConf.moveInfo[4] = em1Row - 1;
newConf.moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
}}
// If the 2 by 1 piece is below the two empty spaces, a new move is generated with the 2
by 1 piece moving up one row.
if ((em1Row+1) < 5 && currentConf.board[em1Row+1][em1Col] == 21 && (em2Row+1)
< 5 && currentConf.board[em2Row+1][em2Col] == 21)
{
newConf = new Board();
// System.out.println("inside check 2 horizon Spaces function 3 if");
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];

// Board is adjusted to reflect the new move.
newConf.board[em1Row][em1Col] = 21;
newConf.board[em2Row][em2Col] = 21;
newConf.board[em1Row+1][em1Col] = 0;
newConf.board[em2Row+1][em2Col] = 0;

// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece: 2 by 1.
newConf.moveInfo[0] = 2;
newConf.moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = em1Row + 1;
newConf.moveInfo[3] = em1Col;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = em1Row;
newConf.moveInfo[5] = em1Col;


if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
```

```
listOfMoves.add(newConf);
}
}
// If the 2 by 1 piece is above the two empty spaces, a new move is generated with the 2
by 1 piece moving down one row.
if ((em1Row-1)>=0 && currentConf.board[em1Row-1][em1Col] == 21 && (em2Row-
1)>=0 && currentConf.board[em2Row-1][em2Col] == 21)
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[em1Row][em1Col] = 21;
newConf.board[em2Row][em2Col] = 21;
newConf.board[em1Row-1][em1Col] = 0;
newConf.board[em2Row-1][em2Col] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece moved: 2 by 1.
newConf.moveInfo[0] = 2;
newConf.moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = em1Row - 1;
newConf.moveInfo[3] = em1Col;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = em1Row;
newConf.moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
} }



// Calls to check1EmptySpace() to generate possible moves for each empty space
separately.
check1EmptySpace(currentConf, em1Row, em1Col);
check1EmptySpace(currentConf, em1Row, em2Col);
}
```

```
////////////////////////////////////////////////////////////////////////////////////////
// The check2VertSpaces() function generates possible moves when the two empty
spaces are
// adjacent vertically.
// It has 5 parameters:
//          currentConf:      Current board configuration.
//          em1Row:           Row of the first empty space.
//          em1Col:           Column of the first empty space.
//          em2Row:           Row of the second empty space.
//          em2Col:           Column of the second empty space.
////////////////////////////////////////////////////////////////////////////////////////
public void check2VertSpaces(Board currentConf, int em1Row, int em1Col, int em2Row,
int em2Col)
{
Board newConf;
boolean isDuplicate;
// If the 2 by 2 piece can move left to the two empty spaces, a new move is generated
with the 2 by 2 moving one column left.
if ((em1Col+1) < 3 && currentConf.board[em1Row][em1Col+1] == 22 && (em2Col+1) <
3 && currentConf.board[em2Row][em2Col+1] == 22)
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[em1Row][em1Col] = 22;
newConf.board[em2Row][em2Col] = 22;
newConf.board[em1Row][em1Col+2] = 0;
newConf.board[em2Row][em2Col+2] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store the dimensions of the piece moved: 2 by 2.
newConf.moveInfo[0] = 2;
newConf.moveInfo[1] = 2;

// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = em1Row;
newConf.moveInfo[3] = em1Col + 1;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = em1Row;
```

```
newConf.moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
}}

// If the 2 by 2 piece can move right to the two empty spaces, a new move is generated
with the 2 by 2 moving one column right.
if ((em1Col-1) > 0 && currentConf.board[em1Row][em1Col-1] == 22 && (em2Col-1) > 0
&& currentConf.board[em2Row][em2Col-1] == 22)
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];

// Board is adjusted to reflect the new move.
newConf.board[em1Row][em1Col] = 22;
newConf.board[em2Row][em2Col] = 22;
newConf.board[em1Row][em1Col-2] = 0;
newConf.board[em2Row][em2Col-2] = 0;

// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 2 and 2 as the dimensions of the piece moved.
newConf.moveInfo[0] = 2;
newConf.moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = em1Row;
newConf.moveInfo[3] = em1Col - 2;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = em1Row;
newConf.moveInfo[5] = em1Col - 1;


if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
```

```
} }

// If the 1 by 2 piece can move left to the two empty spaces, a new move is generated
with the 1 by 2 moving one column left.
if ((em1Col+1)<4 && currentConf.board[em1Row][em1Col+1] == 120 && (em2Col+1)<4
&& currentConf.board[em2Row][em2Col+1] == 12)
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[em1Row][em1Col] = 120;
newConf.board[em2Row][em2Col] = 12;
newConf.board[em1Row][em1Col+1] = 0;
newConf.board[em2Row][em2Col+1] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate){
// moveInfo[0] and moveInfo[1] store 1 and 2 as the dimensions of the piece moved.
newConf.moveInfo[0] = 1;
newConf.moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = em1Row;
newConf.moveInfo[3] = em1Col + 1;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = em1Row;
newConf.moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
}  }

// If the 1 by 2 piece can move right to the two empty spaces, a new move is generated
with the 1 by 2 moving one column right.
if ((em1Col-1)>=0 && currentConf.board[em1Row][em1Col-1] == 120 && (em2Col-
1)>=0 && currentConf.board[em2Row][em2Col-1] == 12)
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
```

```
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[em1Row][em1Col] = 120;
newConf.board[em2Row][em2Col] = 12;
newConf.board[em1Row][em1Col-1] = 0;
newConf.board[em2Row][em2Col-1] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 1 and 2 as the dimensions of the piece moved.
newConf.moveInfo[0] = 1;
newConf.moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = em1Row;
newConf.moveInfo[3] = em1Col - 1;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = em1Row;
newConf.moveInfo[5] = em1Col;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
} }

// Calls to check1EmptySpace() to generate possible moves for each empty space
separately.
check1EmptySpace(currentConf, em1Row, em1Col);
check1EmptySpace(currentConf, em2Row, em2Col);
}


////////////////////////////////////////////////////////////////////////////////////////
// The check1EmptySpace() function generates possible moves for an empty space.
// It has 3 parameters:
//          currentConf:     Current board configuration.
//          emRow:            Row of the empty space.
//          emCol:            Column of the empty space.
////////////////////////////////////////////////////////////////////////////////////////
public void check1EmptySpace(Board currentConf, int emRow, int emCol)
{
Board newConf;
int pieceRow=0;
int pieceCol=0;
```

```
int pieceType=0;
boolean isDuplicate;
// If there is a valid space one row below the empty space.
if ( (emRow+1) < 5 )
{
newConf=new Board();
// Identify piece below empty space.
pieceRow = emRow + 1;
pieceCol = emCol;
pieceType = currentConf.board[pieceRow][pieceCol];
// If the piece below the empty space is a 1 by 1 piece, generate a new move with the 1
by 1 moving up.
if ( pieceType == 11 )
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[emRow][emCol] = pieceType;
newConf.board[pieceRow][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 1 and 1 as the dimensions of the piece moved.
newConf.moveInfo[0] = 1;
newConf.moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = pieceRow;
newConf.moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = emRow;
newConf.moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;


listOfMoves.add(newConf);
}
}
```

```
// If the piece below the empty space is a 1 by 2 piece, generate a new move with the 1
by 2 moving up.
if ( (emRow+2) < 5 && pieceType == 120 )
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[emRow][emCol] = pieceType;
newConf.board[pieceRow][pieceCol] = 12;
newConf.board[pieceRow+1][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 1 and 2 as the dimensions of the piece moved.
newConf.moveInfo[0] = 1;
newConf.moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = pieceRow;
newConf.moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = emRow;
newConf.moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
} } }
// If there is a valid space one row above the empty space.
if ( (emRow-1) >= 0 )
{
newConf=new Board();
// Identify piece above empty space.
pieceRow = emRow - 1;
pieceCol = emCol;

pieceType = currentConf.board[pieceRow][pieceCol];
// If the piece above the empty space is a 1 by 1 piece, generate a new move with the 1
by 1 moving down.
if ( pieceType == 11 )
{
```

```
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[emRow][emCol] = pieceType;
newConf.board[pieceRow][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 1 and 1 as the dimensions of the piece moved.
newConf.moveInfo[0] = 1;
newConf.moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = pieceRow;
newConf.moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = emRow;
newConf.moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
} }
// If the piece above the empty space is a 1 by 2 piece, generate a new move with the 1
by 2 moving down.
if ( (emRow-2) >= 0 && pieceType == 12 )
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[emRow][emCol] = pieceType;

newConf.board[pieceRow][pieceCol] = 120;
newConf.board[pieceRow-1][pieceCol] = 0;
// Call to checkDuplicateConf() board.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
```

```
{
// moveInfo[0] and moveInfo[1] store 1 and 2 as the dimensions of the piece moved.
newConf.moveInfo[0] = 1;
newConf.moveInfo[1] = 2;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = pieceRow - 1;
newConf.moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = pieceRow;
newConf.moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
} } }
// If there is a valid space to the right of the empty space.
if ( (emCol+1) < 4 )
{
newConf=new Board();
// Identify piece to the right of empty space.
pieceRow = emRow;
pieceCol = emCol + 1;
pieceType = currentConf.board[pieceRow][pieceCol];
// If the piece to the right of the empty space is a 1 by 1 piece, generate a new move
with the 1 by 1 moving left.
if ( pieceType == 11 )
{
//  newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[emRow][emCol] = pieceType;
newConf.board[pieceRow][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 1 and 1 as the dimensions of the piece moved.
newConf.moveInfo[0] = 1;
newConf.moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = pieceRow;
```

```
newConf.moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = emRow;
newConf.moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
} }
// If the piece to the right of the empty space is a 2 by 1 piece, generate a new move
with the 2 by 1 moving left.
if ( (emCol+2) < 4 && pieceType == 21 )
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[emRow][emCol] = pieceType;
newConf.board[pieceRow][pieceCol+1] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 2 and 1 as the dimensions of the moved piece.
newConf.moveInfo[0] = 2;
newConf.moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = pieceRow;
newConf.moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = emRow;
newConf.moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;


listOfMoves.add(newConf);
} } }
```

```
// If there is a valid space one column to the left of the empty space.
if ( (emCol-1) >= 0 )
{
newConf=new Board();
// Identify piece to the left of empty space.
pieceRow = emRow;
pieceCol = emCol - 1;
pieceType = currentConf.board[pieceRow][pieceCol];
// If the piece to the left of the empty space is a 1 by 1 piece, generate a new move with
the 1 by 1 moving right.
if ( pieceType == 11 )
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[emRow][emCol] = pieceType;
newConf.board[pieceRow][pieceCol] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 1 and 1 as the dimensions of the piece moved.
newConf.moveInfo[0] = 1;
newConf.moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = pieceRow;
newConf.moveInfo[3] = pieceCol;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = emRow;
newConf.moveInfo[5] = emCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
}
}


// If the piece to the left of the empty space is a 2 by 1 piece, generate a new move with
the 2 by 1 moving right.
if ( (emCol-2) >= 0 && pieceType == 21 )
```

```
{
newConf = new Board();
newConf.moveNumber = currentConf.moveNumber + 1;
newConf.parent = currentConf;
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
newConf.board[i][j] = currentConf.board[i][j];
// Board is adjusted to reflect the new move.
newConf.board[emRow][emCol] = pieceType;
newConf.board[pieceRow][pieceCol-1] = 0;
// Call to checkDuplicateConf() function.
isDuplicate = checkDuplicateConf(newConf);
if (!isDuplicate)
{
// moveInfo[0] and moveInfo[1] store 2 and 1 as the dimensions of the piece moved.
newConf.moveInfo[0] = 2;
newConf.moveInfo[1] = 1;
// moveInfo[2] and moveInfo[3] store the piece's row and column before the move.
newConf.moveInfo[2] = pieceRow;
newConf.moveInfo[3] = pieceCol - 1;
// moveInfo[4] and moveInfo[5] store the piece's row and column after the move.
newConf.moveInfo[4] = pieceRow;
newConf.moveInfo[5] = pieceCol;
if (isHeuristicSearch)
calculateHeuristic(newConf);
else
newConf.heurMeasure = 0;
listOfMoves.add(newConf);
} }  } }


////////////////////////////////////////////////////////////////////////////////
// The checkDuplicateConf() function returns true if newMove is a duplicate of any
// previous configuration, and if newMove has taken more moves to reach this
configuration.
//
// If newMove is a duplicate with fewer moves, then newMove is added to listOfMoves,
// and the deletePath() function is called to delete the less efficient path.
// It has one parameter:
//       newMove:      New configuration.
////////////////////////////////////////////////////////////////////////////////
public boolean checkDuplicateConf(Board newMove)
{
int size_ex1_before_dub=0;
int size_exp2_before_dub =0;
int size_lstmove_before_dub=0;
```

```
boolean flag = true;
Board tempNode=new Board();
// Check if newMove is already contained in listOfMoves.
for (Iterator<Board> iter = listOfMoves.iterator(); iter.hasNext(); )
{
tempNode=iter.next();
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if (newMove.board[i][j] != tempNode.board[i][j])
flag = false;
}
}

if (flag == true)
{
// If newMove reached this configuration in fewer moves, call deletePath() on the
duplicate in listOfMoves.
if ( newMove.moveNumber < tempNode.moveNumber )
{
size_lstmove_before_dub=listOfMoves.size();
iter.remove();
deletePath(tempNode);
SizeMl=SizeMl+(size_lstmove_before_dub - listOfMoves.size());
return false;
}
else
{
return true;
}
}
flag = true;
}

// If this is the breadth-first search, then check the exploredNodes vector.
if (!isHeuristicSearch)
{

// Check if newMove is already contained in exploredNodes.
for (Iterator<Board> iter = exploredNodes.iterator(); iter.hasNext(); )
{
tempNode=iter.next();
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
```

```
{
if (newMove.board[i][j] != tempNode.board[i][j])
flag = false;
}
}

if (flag == true)
{
// If newMove reached this configuration in fewer moves, call deletePath() on the
duplicate in exploredNodes.
if (newMove.moveNumber < tempNode.moveNumber)
{
size_ex1_before_dub=exploredNodes.size();
iter.remove();
deletePath(tempNode);
SizeExp1=SizeExp1+(size_ex1_before_dub-exploredNodes.size());
return false;
}
else
{
return true;
}
}
flag = true;
} }
// Else if this is the heuristic search, check the exploredNodes2 vector.
else
{
// Check if newMove is already contained in exploredNodes2.
tempNode= new Board();
for (Iterator<Board> iter = exploredNodes2.iterator(); iter.hasNext(); )
{

tempNode=iter.next();
for (int i = 0; i < 5; i++)
{
for (int j = 0; j < 4; j++)
{
if (newMove.board[i][j] != tempNode.board[i][j])
flag = false;
}
}
if (flag == true)
{
// If newMove reached this configuration in fewer moves, call deletePath() on the
duplicate in exploredNodes2.
```

```
if (newMove.moveNumber < tempNode.moveNumber)
{
size_exp2_before_dub=exploredNodes2.size();
iter.remove();
deletePath(tempNode);
SizeExp2=SizeExp2+(size_exp2_before_dub - exploredNodes2.size());
return false;
}
else
{
return true;
} }

flag = true;
}  }


return false;

}
////////////////////////////////////////////////////////////////////////////////////
// The deletePath() function deletes less efficient paths when a duplicate configuration
// is found by the checkDuplicateConf() function.
// It has one parameter:
//        deleteNode:  Node to be deleted.
////////////////////////////////////////////////////////////////////////////////////
public void deletePath(Board deleteNode)
{
Board TempNode22;
// Check listOfMoves for nodes which are descendants of deleteNode.
if ( !listOfMoves.isEmpty())
{
TempNode22= new Board();
for (Iterator<Board> iter = listOfMoves.iterator(); iter.hasNext(); )
{
TempNode22=iter.next();
if ( TempNode22.parent != null && TempNode22.parent == deleteNode )
{
iter.remove();
deletePath(TempNode22);
}
}
}

// If this is the breadth-first search, check exploredNodes for nodes which are
descendants of deleteNode.
```

```
if ( !isHeuristicSearch && !exploredNodes.isEmpty())
{
for(Board Lst:exploredNodes)
{
// TempNode22=iter.next();
if ( Lst.parent != null && Lst.parent == deleteNode )
{
exploredNodes.remove(Lst);
deletePath(Lst);
} } }

// If this is the heuristic search, check exploredNodes2 for nodes which are descendants
of deleteNode.
if ( isHeuristicSearch && !exploredNodes2.isEmpty() )
{
for (Iterator<Board> iter = listOfMoves.iterator(); iter.hasNext(); )
{
TempNode22=iter.next();
// for (Board Lst:exploredNodes2)

if ( TempNode22.parent != null && TempNode22.parent == deleteNode )
{
//if(!exploredNodes2.isEmpty())
iter.remove();
deletePath(TempNode22);
}

}
}

}

////////////////////////////////////////////////////////////////////////////////
// The printSolution() function prints information about a configuration and the sequence
// of moves leading to this configuration.
// It has two parameters:
//       printNode:   Configuration to be printed.
//       printRequired: A boolean flag set to true when printing the entire sequence of
//                      moves, and set to false when printing only the configuration.
////////////////////////////////////////////////////////////////////////////////
public  void printSolution(Board printNode, boolean printRequired) {
Board tempNode=new Board();
int solutionSize;
try {
fstream = new FileWriter("output.txt");
out= new BufferedWriter(fstream);
```

```
// Print out the configuration.
out.write("Final configuration: ");
out.newLine();
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
{
if ( printNode.board[i][j] == 11 ){
out.write("The 1 by 1 piece is located at row "+i+" and column " +j +" ");
out.newLine();}
else if ( printNode.board[i][j] == 21 && (j+1) < 4 && printNode.board[i][j+1] == 21 ){
out.write("The 2 by 1 piece is located at row "+i+ " and column "+j+" ");
out.newLine();}
else if ( printNode.board[i][j] == 120 ){
out.write("The 1 by 2 piece is located at row "+i+" and column "+j);
out.newLine();}
else if ( printNode.board[i][j] == 22 && (i+1) < 5 && printNode.board[i+1][j] == 22
&& (j+1) < 4 && printNode.board[i][j+1] == 22 ){
out.write("The 2 by 2 piece is located at row "+i+" and column "+j);
out.newLine();}
}
out.write( "\n\n");
out.newLine();
// If the printRequired flag is set to true, print out the entire sequence of moves.
if (printRequired)
{
solutionSize = solutionList.size();

out.write( "At the end of the program, the size of the solutionList is "+ solutionSize+
"\n\n");
out.newLine();
out.write("\n\nThe sequence of moves leading to this configuration is:\n\n");
out.newLine();
// For-loop prints out the sequence of moves.
for (int i = 1; i < solutionSize - 1; i++)
{
out.write("Move "+solutionList.get(i).moveNumber + ": The ");
//out.newLine();
out.write( solutionList.get(i).moveInfo[0] + " by "+solutionList.get(i).moveInfo[1]);
//out.newLine();
out.write(" piece was moved from row ");
// out.newLine();

// If the moveNumber is 40 or less, moveInfo[2] and moveInfo[3] are the starting row and
column for the piece,
// and moveInfo[4] and moveInfo[5] are the new row and column after the move.
if (solutionList.get(i).moveNumber <= 40)
```

```
{
out.write(""+solutionList.get(i).moveInfo[2]+"");
//out.newLine();
out.write( " and column " +solutionList.get(i).moveInfo[3]);
//out.newLine();
out.write( " to row "+ solutionList.get(i).moveInfo[4]);
//out.newLine();
out.write( " and column " + solutionList.get(i).moveInfo[5]);
out.newLine();
//outfile << endl << endl;
}

// Else if moveNumber is more than 40, moveInfo[4] and moveInfo[5] are the starting
row and column for the piece,
// and moveInfo[2] and moveInfo[3] are the new row and column after the move.
else
{
out.write("- "+solutionList.get(i).moveInfo[4]);
//out.newLine();
out.write( " and column " + solutionList.get(i).moveInfo[5]);
//out.newLine();
out.write( " to row " +solutionList.get(i).moveInfo[2]);
//out.newLine();
out.write( " and column " + solutionList.get(i).moveInfo[3]);
out.newLine();
// outfile << endl << endl;
}
}
out.write("\n\n");
out.newLine();
solutionList.clear();
}
out.close();
}catch(IOException e){       System.out.println("There was a problem:" + e);
}
}
```

```
////////////////////////////////////////////////////////////////////////////////
// The calculateHeuristic() function assigns a heuristic estimate of the new move's
distance
// to the solution.
// It has one parameter:
//        newConf:     new board.
////////////////////////////////////////////////////////////////////////////////
public  void calculateHeuristic(Board newConf)
{
int squareRow=0;
int squareCol=0;
int em1Row=0;
int  em1Col=0;
int em2Row=0;
int em2Col=0;
int firstEmIndex=0;
int pieceRow=0;
int pieceCol=0;
boolean isFirstEmpty = true;
Board tempNode=new Board();

// For-loops find the row and column of the 2 by 2 square in this configuration.
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
{
if (newConf.board[i][j] == 22 && (i+1) < 5 && newConf.board[i+1][j] == 22 && (j+1) < 4
&& newConf.board[i][j+1] == 22)
{
squareRow = i;
squareCol = j;
break;
}
}

// Heuristic is assigned by taking the 2 by 2 square's current row: Row 0 is ideal to reach
the starting position.
newConf.heurMeasure = squareRow;
// For-loops find the empty spaces in the new board.
for (int i = 0; i < 5; i++)
for (int j = 0; j < 4; j++)
{
if ( newConf.board[i][j] == 0 )
{
// If this is the first empty space, the row and column are assigned to em1Row and
em1Col.
if (isFirstEmpty)
```

```
{
em1Row = i;
em1Col = j;
isFirstEmpty = false;
}

// Else if this is the second empty space, the row and column are assigned to em2Row
and em2Col.
else
{
em2Row = i;
em2Col = j;
}
}
}

isFirstEmpty = true;

// If the 2 by 2 square can move up one row, improve its heuristic score by subtracting 2.
if ( (squareRow-1) >= 0 && em1Row == (squareRow-1) && em2Row == (squareRow-1)
&& em1Col == squareCol
&& (squareCol+1) < 4 && em2Col == (squareCol+1) )
newConf.heurMeasure -= 2.0;

// Else if it can move down one row, improve its heuristic score by subtracting 0.5.
else if ( (squareRow+2) < 5 && em1Row == (squareRow+2) && em2Row ==
(squareRow+2) && em1Col == squareCol
&& (squareCol+1) < 4 && em2Col == (squareCol+1) )
newConf.heurMeasure -= 0.5;
// Else if it can move right one column, improve its heuristic score by subtracting 1.
else if ( (squareRow+1) < 5 && em1Row == squareRow && em2Row == (squareRow+1)
&& (squareCol+2) < 4
&& em1Col == (squareCol+2) && em2Col == (squareCol+2) )
newConf.heurMeasure -= 1;
// Else if it can move left one column, improve its heuristic score by subtracting 1.
else if ( (squareRow+1) < 5 && em1Row == squareRow && em2Row == (squareRow+1)
&& (squareCol-1) >= 0
&& em1Col == (squareCol-1) && em2Col == (squareCol-1) )
newConf.heurMeasure -= 1;
// Else if there are pieces above the 2 by 2, check possible moves for these pieces.
else if ( (squareRow-1) >= 0 )
{
pieceRow = squareRow - 1;

for (int i = 0; i < 2; i++)
{
```

```
if (i == 0)
pieceCol = squareCol;
else if (i == 1)
pieceCol = squareCol + 1;

// If the piece above the 2 by 2 is a 1 by 1 piece, check possible moves for the 1 by 1.
if ( pieceCol < 4 && newConf.board[pieceRow][pieceCol] == 11 )
{
// If the 1 by 1 can move right or left, the heuristic estimate is improved by subtracting
0.5.
if ( em1Row == pieceRow && (pieceCol+1) < 4 && em1Col == (pieceCol+1) )
newConf.heurMeasure -= 0.5;
else if ( em1Row == pieceRow && (pieceCol-1) >= 0 && em1Col == (pieceCol-1) )
newConf.heurMeasure -= 0.5;
else if ( em2Row == pieceRow && (pieceCol+1) < 4 && em2Col == (pieceCol+1) )
newConf.heurMeasure -= 0.5;
else if ( em2Row == pieceRow && (pieceCol-1) >= 0 && em2Col == (pieceCol-1) )
newConf.heurMeasure -= 0.5;
// If the 1 by 1 can move up, the heuristic estimate is improved by subtracting 0.5.
if ( (pieceRow-1) >= 0 && em1Col == pieceCol && em1Row == (pieceRow-1) )
newConf.heurMeasure -= 0.5;
else if ( em2Col == pieceCol && (pieceRow-1) >= 0 && em2Row == (pieceRow-1) )
newConf.heurMeasure -= 0.5;

}

// If the piece above the 2 by 2 is a 1 by 2 piece, check possible moves for the 1 by 2.
else if ( pieceCol < 4 && newConf.board[pieceRow][pieceCol] == 120 )
{

// If the 1 by 2 can move up, the heuristic estimate is improved by subtracting 0.5.
if ( em1Col == pieceCol && (pieceRow-1) >= 0 && em1Row == (pieceRow-1) )
newConf.heurMeasure -= 0.5;
else if ( em2Col == pieceCol && (pieceRow-1) >= 0 && em2Row == (pieceRow-1) )
newConf.heurMeasure -= 0.5;

// If the 1 by 2 can move left or right, the heuristic estimate is improved by subtracting
0.5.
if ( em1Row == pieceRow && (pieceRow+1) < 5 && em2Row == (pieceRow+1) &&
(pieceCol+1) < 4
&& em1Col == (pieceCol+1) && em2Col == (pieceCol+1) )
newConf.heurMeasure -= 0.5;

else if ( em1Row == pieceRow && (pieceRow+1) < 5 && em2Row == (pieceRow+1) &&
(pieceCol-1) >= 0
&& em1Col == (pieceCol-1) && em2Col == (pieceCol-1) )
```

```
newConf.heurMeasure -= 0.5;

}


// If the piece above the 2 by 2 is a 2 by 1 piece, check possible moves for the 2 by 1.
else if ( pieceCol < 4 && newConf.board[pieceRow][pieceCol] == 21 )
{

// If the 2 by 1 can move up, the heuristic estimate is improved by subtracting 1.
if ( (pieceRow-1) >= 0 && em1Row == (pieceRow-1) && em2Row == (pieceRow-1) &&
em1Col == pieceCol
&& (pieceCol+1) < 4 && em2Col == (pieceCol+1) )
newConf.heurMeasure -= 1;
else
{
// If the 2 by 1 can move left or right, the heuristic estimate is improved by subtracting
0.5.
if ( em1Row == pieceRow && (pieceCol+2) < 4 && em1Col == (pieceCol+2) )
newConf.heurMeasure -= 0.5;
else if ( em1Row == pieceRow && (pieceCol-1) >= 0 && em1Col == (pieceCol-1) )
newConf.heurMeasure -= 0.5;

if ( em2Row == pieceRow && (pieceCol+2) < 4 && em2Col == (pieceCol+2) )
newConf.heurMeasure -= 0.5;
else if ( em2Row == pieceRow && (pieceCol-1) >= 0 && em2Col == (pieceCol-1) )
newConf.heurMeasure -= 0.5;
}
}
}       // For-loop's closing brace
}

// Multiply the heurMeasure by 100 to make it an integer value.
newConf.heurMeasure *= 100.0;
}
```

```
////////////////////////////////////////////////////////////////////////////////////
// The buildSolutionList() function combines the two paths found by the bidirectional
search
// into a single path from the start state to a solution.
// It has two parameters:
//        finalNode:   Last node explored by the heuristic search.
//        move40:      Node in move40List which is identical to finalNode.
////////////////////////////////////////////////////////////////////////////////////
public void buildSolutionList(Board finalNode, Board move40) {

Board tempNode=new Board();

int solutionSize;

int count;
List <Board>  tempList= new LinkedList();

// Build the path from move40 back to the start state.
tempList.add(move40);

tempNode = move40.parent;

while (tempNode != null)
{
tempList.add(tempNode);

tempNode = tempNode.parent;
}

// Transfer this path from tempList to solutionList.
for (int i = tempList.size() - 1; i >= 0; i--)
{
solutionList.add(tempList.get(i));
}

// Assign 41 as the moveNumber for finalNode, and build the path from finalNode to the
solution state.
finalNode.moveNumber = 41;
solutionList.add(finalNode);
tempNode = finalNode.parent;
count = 2;
while (tempNode != null)
{
tempNode.moveNumber = 40 + count;
solutionList.add(tempNode);
```

```
tempNode = tempNode.parent;
count++;
}
// out.write( "The size of the solutionList is " + solutionList.size()+ ".\n\n");
solutionSize = solutionList.size();
// Call to printSolution() to print out the solution path.
tempNode = solutionList.get(solutionSize - 1);

printSolution(tempNode, true);
}
}
```

Appendix C

Java Implementations of Fifteen Puzzle

```
/*
* To change this template, choose Tools | Templates
* and open the template in the editor.
*/

package puzzlesolver;
import java.util.*;
/**
* Main function for All fifteen Puzzles
* @author kose
*/
public class Main {

/**
* @param args the command line arguments
* /**
* File: Main.java
* Command line application
*
* Some Goal example configurations:
*    {8,0,2,3,10,4,5,7,1,9,6,11,12,13,14,15};
*    {8,0,2,3,10,4,5,7,1,9,6,11,12,13,14,15};
*    {14,13,15,7,11,12,9,5,6,0,2,1,4,8,10,3};
*    {8,2,6,7,10,3,15,4,13,7,0,11,1,12,9,14};
*    {14,13,15,7,11,12,9,5,6,0,2,1,4,8,10,3};
*/

public static void main(String[] args) {
// TODO code application logic here
// int [] b=new int[2];
/*
* the Start board configuration is
* 0-1-2-3
* 4-5-6-7
* 8-9-10-11
* 12-13-14-15
* or
* {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
*/
LinkedList<Node> IDAList = new LinkedList<Node>();
LinkedList<Node> IDAList1 = new LinkedList<Node>();
```

```
LinkedList<Node> AStartList = new LinkedList<Node>();
LinkedList<Node> IndexBFSList = new LinkedList<Node>();
LinkedList<Node> BFSOpen = new LinkedList<Node>();
LinkedList<Node> BFSClose = new LinkedList<Node>();
LinkedList<Node> AStarOpen = new LinkedList<Node>();
LinkedList<Node> AStarClose = new LinkedList<Node>();
LinkedList<Node> L = new LinkedList<Node>();
//HashMap <Integer,Node> hm1 = new HashMap <Integer,Node>();
// HashMap <Long,Node> hm2  = new HashMap <Long,Node>();
Heuristic H= new Heuristic();
Algorithms Alg=new Algorithms();
Search S=new Search();
Node Start= new Node();
Node G=new Node();
// Node Goal= new Node();
Node Pilot= new Node();
//AStarNode Start1= new AStarNode();
int Depth=0;
int SolutionDepth=0;

Start.board = new int [] {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int[] Goal = new int [] {4,5,7,2,9,14,12,13,0,3,6,11,8,1,15,10};
G.board = new int [] {14,13,15,7,11,12,9,5,6,0,2,1,4,8,10,3};

Start.h=H.H(Start.board, Goal);
System.out.println("Start Huristic value :  "+Start.h);
// Start1.AStar(Start.board,Goal);
// Alg.BFS(Start,OpenBFS,Close);
IndexBFSList.clear();
AStartList.clear();
IDAList.clear();
BFSOpen.clear();
BFSClose.clear();

/*
*BFS Outputs
*/

BFSOpen.add(Start);
System.out.println("BFS Solution Depth :"+Alg.BFS(BFSOpen, BFSClose, G.board));
System.out.println("Open List Size :"+BFSOpen.size());
System.out.println("Close List Size :"+BFSClose.size());

/*
*IBFS Outputs
*/
```

```
IndexBFSList.add(Start);
System.out.println("BFS Solution Depth :"+Alg.IBFS(IndexBFSList, G.board));
System.out.println("Open List Size :"+IndexBFSList.size());
System.out.println("Close List Size :"+BFSClose.size());

/*
*IA*  Outputs
*/
System.out.println("IAStar Solution Depth :"+Alg.IAStar(Start,AStartList,Goal));
System.out.println("IAStart AStartList List Size :"+AStartList.size());

/*
*A* Outputs
*/
System.out.println("AStar Solution Depth :"+Alg.AStar(Start,
AStarOpen,AStarClose,Goal));
System.out.println("AStart OPEN List Size :"+AStarOpen.size());
System.out.println("AStart CLOSE List Size :"+AStarClose.size());

}
}
/*
* To change this template, choose Tools | Templates
* and open the template in the editor.
*/
package puzzlesolver;

/**
*Node Representation of Fifteen Puzzle
* @author kose
* Move ways, 0->left, 1-> right, 2-> down and 3->up
*
*/
public class Node {
int[] board;
int h;
int MoveWay;
long MoveIndex;
int DupFlag;
int depth;
public  Node(){
MoveWay=4; MoveIndex=0; DupFlag=4; depth =0;
board = new int[16];
}
```

```
}

/*
* To change this template, choose Tools | Templates
* and open the template in the editor.
*/

package puzzlesolver;
import java.util.*;

/**
*
* @author kose
* IBFS,BFS, IA*, A* and IDA* Algorithm implementations
*/
public class Algorithms {

Search S=new Search();
BFSMoves BFS=new BFSMoves();
Heuristic H= new Heuristic();
int []Start = new int [] {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int Depth=0;
/*
* Indexed First Search Algorithm
*/
public int IBFS(LinkedList<Node> Open1, int[] Goal){

Node N=new Node();
int depth=0;
while(N.depth<14){
Node NewConf=new Node();
N= Open1.removeFirst();
int Empty=S.FindEmpty(N.board);


/*
LEFT MOVE
*/
if((N.MoveWay!=1) && (N.DupFlag!=0) && !(Empty ==0 || Empty ==4 || Empty ==8
||Empty ==12 )){
NewConf=S.Left(N,Empty);
if(S.CheckSolution(NewConf,Goal))
return NewConf.depth;
if (!S.CheckDuplication(NewConf,Open1)){
Open1.add(NewConf);
}
```

```
}
/*
RIGHT MOVE
*/
if((N.MoveWay!=0) && (N.DupFlag!=1) &&!(Empty ==3 || Empty ==7 || Empty ==11
||Empty ==15 )){

NewConf=S.Right(N,Empty);
if(S.CheckSolution(NewConf,Goal))
return NewConf.depth;
if (!S.CheckDuplication(NewConf, Open1)){
Open1.add(NewConf);
}
}
/*
DOWN MOVE
*/
if((N.MoveWay!=3) && (N.DupFlag!=2)&&(Empty <12 ))
{
NewConf=S.Down(N,Empty);
if(S.CheckSolution(NewConf,Goal))
return NewConf.depth;
if (!S.CheckDuplication(NewConf, Open1)){
Open1.add(NewConf);
}
}
/*
UP MOVE
*/
if((N.MoveWay!=2) && (N.DupFlag!=3) && (Empty >3)){
NewConf= S.Up(N,Empty);
if(S.CheckSolution(NewConf,Goal))
return NewConf.depth;
if (!S.CheckDuplication(NewConf, Open1)){
Open1.add(NewConf);
}
}
}
return 0;
}

/*
*Breadth First Search Algorithm
*/
public int BFS(LinkedList<Node> Open, LinkedList<Node>Close, int [] Goal){
Heuristic H= new Heuristic();
```

```
Node N=new Node();//Current Configuration

while (N.depth<11)
{
Node NewConf=new Node();
N=Open.removeFirst();
int Empty=S.FindEmpty(N.board);
Close.add(N);

/*
LEFT MOVE
*/
if(!(Empty ==0 || Empty ==4 || Empty ==8 ||Empty ==12 )){
NewConf=BFS.Left(N,Empty);
if(S.CheckSolution(NewConf,Goal))
return NewConf.depth;
if (!BFS.CheckDuplicationBFS(NewConf,Close))
Open.add(NewConf);
}
/*
RIGHT MOVE
*/
if(!(Empty ==3 || Empty ==7 || Empty ==11 ||Empty ==15 )){

NewConf=BFS.Right(N,Empty);

if(S.CheckSolution(NewConf,Goal))
return NewConf.depth;
if (!BFS.CheckDuplicationBFS(NewConf, Close))
Open.add(NewConf);

}
/*
DOWN MOVE
*/
if(Empty <12)
{
NewConf=BFS.Down(N,Empty);

if(S.CheckSolution(NewConf,Goal))
return NewConf.depth;
if (!BFS.CheckDuplicationBFS(NewConf, Close))
Open.add(NewConf);

}
```

```
/*
UP MOVE
*/
if(Empty >3){
NewConf= BFS.Up(N,Empty);

if(S.CheckSolution(NewConf,Goal))
return NewConf.depth;
if (!BFS.CheckDuplicationBFS(NewConf, Close))
Open.add(NewConf);

}
}
System.out.println("the depth is "+N.depth);
return 0;
}

/*
* Indexed AStart Algorithm
*/

public int IAStar(Node Start,LinkedList <Node> NodeList,  int [] Goal){

Node N=new Node();
Node NewNode=new Node();
//Search S=new Search();
N.h=H.H(Start.board,Goal);
System.arraycopy(Start.board, 0, N.board, 0, Start.board.length);
S.AStarAdd(NodeList,N);
long NodeExpanded=0;
while(N.depth<56){
N=NodeList.remove();
NodeExpanded++;
int Empty=S.FindEmpty(N.board);

/*
LEFT MOVE
*/

if((N.MoveWay!=1) && (N.DupFlag!=0) && !(Empty ==0 || Empty ==4 || Empty ==8
||Empty ==12 )){
NewNode=S.Left(N,Empty); // Try To Move left
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(S.CheckSolution(NewNode,Goal)){
System.out.println("Node Expanded: "+NodeExpanded);
return NewNode.depth;
```

```
}
else {
// if (!S.CheckDuplication(NewNode,NodeList))
S.AStarAdd(NodeList,NewNode);
}

}
/*
RIGHT MOVE
*/
if((N.MoveWay!=0) && (N.DupFlag!=1) &&!(Empty ==3 || Empty ==7 || Empty ==11
||Empty ==15 )){
NewNode=S.Right(N,Empty);// Try To Move     the blank space Righ
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(S.CheckSolution(NewNode,Goal)){
System.out.println("Node Expanded: "+NodeExpanded);
return NewNode.depth;
}
else{
S.AStarAdd(NodeList,NewNode);
}
}

/*
DOWN MOVE
*/
if((N.MoveWay!=3) && (N.DupFlag!=2)&&(Empty <12 )){
//System.out.println("in DOWN");
NewNode=S.Down(N,Empty);// Try To Move the blank space Down
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(S.CheckSolution(NewNode,Goal)){
System.out.println("Node Expanded: "+NodeExpanded);
return NewNode.depth;
}

else{
// if (!S.CheckDuplication(NewNode,NodeList))
S.AStarAdd(NodeList,NewNode);
}

}

/*
UP MOVE
*/
if((N.MoveWay!=2) && (N.DupFlag!=3) && (Empty >3))
```

```
{
// System.out.println("in UP");
NewNode=S.Up(N,Empty); //  Try To Move the blank space up
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(S.CheckSolution(NewNode,Goal)){
System.out.println("Node Expanded: "+NodeExpanded);
return NewNode.depth;
} else{
//if (!S.CheckDuplication(NewNode,NodeList))
S.AStarAdd(NodeList,NewNode);
}
}

}
System.out.println("Node Expanded: "+NodeExpanded);
return 0;
}
/*
* AStart Algorithm
*/
public int AStar(Node Start,LinkedList <Node> Open, LinkedList <Node> Close,  int []
Goal){

Node N=new Node();
Node NewNode=new Node();
N.h=H.H(Start.board,Goal);
System.arraycopy(Start.board, 0, N.board, 0, Start.board.length);
S.AStarAdd(Open,N);
long NodeExpanded=0;
while(N.depth<56){
N=Open.remove();
Close.add(N);
NodeExpanded++;
int Empty=S.FindEmpty(N.board);
/*
LEFT MOVE
*/
if((N.MoveWay!=1) && (N.DupFlag!=0) && !(Empty ==0 || Empty ==4 || Empty ==8
||Empty ==12 )){
// System.out.println("in LEFT");
NewNode=S.Left(N,Empty); // Try To Move left
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(S.CheckSolution(NewNode,Goal)){
System.out.println("Node Expanded: "+NodeExpanded);
return NewNode.depth;
}
```

```
else

S.AStarAdd(Open,NewNode);
}
/*
RIGHT MOVE
*/
if((N.MoveWay!=0) && (N.DupFlag!=1) &&!(Empty ==3 || Empty ==7 || Empty ==11
||Empty ==15 )){
NewNode=S.Right(N,Empty);// Try To Move      the blank space Righ
//System.out.println("in RIGHT");
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(S.CheckSolution(NewNode,Goal)){
System.out.println("Node Expanded: "+NodeExpanded);
return NewNode.depth;
}
else
S.AStarAdd(Open,NewNode);
}

/*
DOWN MOVE
*/
if((N.MoveWay!=3) && (N.DupFlag!=2)&&(Empty <12 )){
//System.out.println("in DOWN");
NewNode=S.Down(N,Empty);// Try To Move the blank space Down
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(S.CheckSolution(NewNode,Goal)){
System.out.println("Node Expanded: "+NodeExpanded);
return NewNode.depth;
}

else
S.AStarAdd(Open,NewNode);
}

/*
UP MOVE
*/
if((N.MoveWay!=2) && (N.DupFlag!=3) && (Empty >3)) {
// System.out.println("in UP");
NewNode=S.Up(N,Empty); //  Try To Move the blank space up
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(S.CheckSolution(NewNode,Goal)){
System.out.println("Node Expanded: "+NodeExpanded);
return NewNode.depth;
```

```
} else
S.AStarAdd(Open,NewNode);
}
}
System.out.println("Node Expanded: "+NodeExpanded);
return 0;
}

/*
* IDA* Algorithm
*/
public int IDAStart(Node Start, LinkedList <Node> NodeList, int T, int StartDist, int D, int[]
Goal ){
int T1=0;
int Min=200;
Node N =new Node();
Node NewNode=new Node();
boolean r=false;
if(NodeList.isEmpty())
NodeList.add(0,Start);

while(!NodeList.isEmpty()){
N=NodeList.removeFirst();
int Empty=S.FindEmpty(N.board);
/*
LEFT MOVE
*/
if((N.MoveWay!=1) && (N.DupFlag!=0) && !(Empty ==0 || Empty ==4 || Empty ==8
||Empty ==12 )){
NewNode= S.Left(N, Empty);
if(S.CheckSolution(NewNode,Goal)){
r=true;
Depth=NewNode.depth;
NodeList.addFirst(N);
break;
}
NewNode.h=H.H(NewNode.board, Goal)+ NewNode.depth;
if(NewNode.h+StartDist>=D){
r=true;
//System.out.println("CUT1:"+NewNode.h);
Depth=0;
break;
}

Min=S.IDAStartAdd(NewNode,Min, T, NodeList);
}
```

```
/*
RIGHT MOVE
*/
if((N.MoveWay!=0) && (N.DupFlag!=1) &&!(Empty ==3 || Empty ==7 || Empty ==11
||Empty ==15 ))
{
NewNode= S.Right(N, Empty);
if(S.CheckSolution(NewNode,Goal)){
r=true;
Depth=NewNode.depth;
NodeList.addFirst(N);
break;
}
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;

Min=S.IDAStartAdd(NewNode,Min, T, NodeList);
}

/*
DOWN MOVE
*/
if((N.MoveWay!=3) && (N.DupFlag!=2)&&(Empty <12 ))
{
NewNode= S.Down(N, Empty);
if(S.CheckSolution(NewNode,Goal)){
Depth=NewNode.depth;
NodeList.addFirst(N);
r=true;
break;
}
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(NewNode.h+StartDist>=D){
r=true;
//  System.out.println("CUT1:"+NewNode.h);
Depth=0;
break;
}
Min=S.IDAStartAdd(NewNode,Min, T, NodeList);
}
/*
UP MOVE
*/
if((N.MoveWay!=2) && (N.DupFlag!=3) && (Empty >3))
{
NewNode= S.Up(N, Empty);
if(S.CheckSolution(NewNode,Goal)) {
```

```
Depth=NewNode.depth;
NodeList.addFirst(N);
r=true;
break;
}
NewNode.h=H.H(NewNode.board,Goal)+ NewNode.depth;
if(NewNode.h+StartDist>=D){
r=true;
// System.out.println("CUT1:"+NewNode.h);
Depth=0;
break;
}
Min=S.IDAStartAdd(NewNode,Min, T, NodeList);
}
if(Min>=T)
T1=Min;
// System.out.println("TempMin "+Min);
}

if(r)
return Depth;
IDAStart(Start,NodeList,T1,StartDist, D, Goal);
return Depth;

}
}


/*
* To change this template, choose Tools | Templates
* and open the template in the editor.
*/

package puzzlesolver;
import java.util.*;
import java.io.*;
import java.lang.Math.*;

/**
* All function for Generating Nodes and all other functions for IBFS, IA*, A*, IDA
algorithms
* @author kose
*/
public class Search {

FileWriter fstream;
BufferedWriter out;
```

```
Heuristic H= new Heuristic();

// generate Left Move

public Node  Left(Node N, int Empty){
Node NewNode = new Node();
Heuristic H= new Heuristic();
System.arraycopy(N.board, 0, NewNode.board, 0, N.board.length);
NewNode.board[Empty] = N.board[Empty-1] ;
NewNode.board[Empty-1]=0;
NewNode.MoveWay=0;
NewNode.MoveIndex=N.MoveIndex*4;
NewNode.depth=N.depth+1;
return NewNode;
}
//Generate Right move
public Node Right(Node N,int Empty){

Node newConf=new Node();
System.arraycopy(N.board, 0, newConf.board, 0, 16);
newConf.board[Empty] = N.board[Empty+1] ;
newConf.board[Empty+1]=0;
newConf.MoveWay=1;
newConf.MoveIndex=N.MoveIndex*4+1;
newConf.depth=N.depth+1;
return newConf;
}

// Generate Down Move
public Node Down(Node N, int Empty){
Node NewNode=new Node();
System.arraycopy(N.board, 0, NewNode.board, 0, 16);
NewNode.board[Empty] = N.board[Empty+4] ;
NewNode.board[Empty+4]=0;
NewNode.MoveWay=2;
NewNode.MoveIndex=N.MoveIndex*4+2;
NewNode.depth=N.depth+1;
return NewNode;
}
//Generate Up Move
public Node Up(Node N,int Empty){
// Move the Empty Tile to up.
Node NewNode=new Node();
System.arraycopy(N.board, 0, NewNode.board, 0, 16);
NewNode.board[Empty] = N.board[Empty-4];
NewNode.board[Empty-4]=0;
```

```
NewNode.MoveWay=3;
NewNode.MoveIndex=N.MoveIndex*4+3;
NewNode.depth=N.depth+1;
return NewNode;
}

// Add anode to the IDA* List
public int IDAStartAdd(Node N, int m, int T, LinkedList <Node> NodeList){
if(N.h<=T)
NodeList.addFirst(N);
else {
if(N.h<m)
m= N.h;
}

return m;
}

/*
*Add  a Node to the biggenting of A* and IA*
*/

// Binary Search
public void AStarAdd(LinkedList<Node>NodeList, Node b) {
int middle=0;
if (NodeList.size() == 0) {
NodeList.add(0,b);
return;
}
int low = 0;
int high = NodeList.size()-1;
while(low <= high) {
middle = (low+high) /2;
Node Temp=NodeList.get(middle);
if (b.h> Temp.h){
low = middle +1;
} else if (b.h< Temp.h){
high = middle -1;
} else {          // The element has been found
NodeList.add(middle,b);
return;
}
}
NodeList.add(middle,b);
return;
}
```

```
//Add I Node To IDA* List
public int IDAStartAdd(LinkedList<Node>NodeList,int Min, Node NewNode, int T){

if(NewNode.h<=T)
NodeList.addFirst( NewNode);
else{
if(NewNode.h<Min)
Min = NewNode.h;
}
return Min;
}

// fined the didtance between two nodes

public Node NodeDist(LinkedList <Node> IDAStarList){

Node N=IDAStarList.getFirst();
Node NewNode=new Node();
System.out.println("IDASatr List Last Node ");
int z=0;
// System.arraycopy(currentConf.board, 0, x.board, 0, 16);
for(int j=0;j<N.board.length;j++){
if(N.board[j]==0)
z=j;

}
switch (N.MoveWay){
case 0:
NewNode= Right(N,z);
break;
case 1:
NewNode= Left(N,z);
break;
case 2:
NewNode=Up(N,z);
break;
case 3:
NewNode=Down(N,z);
break;

}
return NewNode;
}
/*
* Find Empty Tile Function
*/
```

```
public int FindEmpty( int[] b)
{
for (int i = 0; i<b.length; i++)
{
if (b[i] == 0)
return i;
}
return 0;

}
/*
* Check Soulution Function
*/

public boolean CheckSolution( Node NewNode, int []Goal){
if(Arrays.equals(NewNode.board, Goal)) {
System.out.println("Solution Exists at Level "+NewNode.depth+" of the tree");
System.out.println("Solution is "+NewNode.MoveIndex+" of the tree");
System.out.println("Solution un base 4 "+Long.toString(NewNode.MoveIndex,4));
//System.out.println("the sizzzz of List With Duplication: "+ NodeList.size());
//System.exit(0);
return true;
}
return false;
}

public boolean CheckListSolution( Node NewNode, LinkedList <Node> Front){
ListIterator <Node>Litr = Front.listIterator();
while(Litr.hasNext()){
Node N = Litr.next();
if(Arrays.equals(NewNode.board, N.board)) {
System.out.println("Solution Exists at Level "+NewNode.depth+" of the tree");
// System.out.println("Solution is "+NewNode.MoveIndex+" of the tree");
//System.out.println("Solution un base 4 "+Long.toString(NewNode.MoveIndex,4));
//System.out.println("the sizzzz of List With Duplication: "+ NodeList.size());
//System.exit(0);
return true;
}
}
return false;
}
public Node PilotNode(LinkedList <Node> Open1,LinkedList <Node> L, int[] Goal){
Node Pilot= new Node();
int MinH=2000;
int Max=0;
ListIterator <Node>Listitr = Open1.listIterator();
```

```
while(Listitr.hasNext()) {
Node N = Listitr.next();
N.h=H.H(Goal, N.board);
Listitr.set(N);
AStarAdd(L,N);
Listitr.remove();
if(MinH>N.h){
MinH=N.h;
Pilot=N;
// i=Listitr.nextIndex();
}
if(Max<N.h)
Max=N.h;
}
System.out.println("the max h:"+Max);
return Pilot;
}

/*
* Removenode For IBFS
*/
public void RemoveNodes(LinkedList <Node> Front, int MinDist){

ListIterator <Node>Listitr = Front.listIterator();
while(Listitr.hasNext()){
Node N = Listitr.next();
if(N.h>=MinDist)
Listitr.remove();
}

}

//IBFS and IA* Algorithms checkduplication Function

public boolean CheckDuplication(Node dup,LinkedList <Node>Lst){

if(Lst.isEmpty())
return false;
ListIterator <Node>Litr = Lst.listIterator();
boolean flag1=false;
search1:
while(Litr.hasNext()) {
boolean flag=true;
Node N = Litr.next();
search:
for(int i: dup.board){
```

```
if(dup.board[i]!=N.board[i]){
flag=false;
break search;
}
}
if(flag){
switch (dup.MoveWay){
case 0:
N.DupFlag=1;
break;
case 1:
N.DupFlag=0;
break;
case 2:
N.DupFlag=3;
break;
case 3:
N.DupFlag=2;
break;

}
Litr.set(N);
flag1=true;
break  search1;
}
}
return flag1;
}

// Binary Search
/*
* Calculate the diferneces between two Indecies.
*/
public int IndexDifference(long X,long Y){

String Str1=Long.toString(X,4);
String Str2=Long.toString(Y,4);
System.out.println("Pilot.MoveIndex   "+Str1);
System.out.println("L.getFirst().MoveIndex  "+Str2);
int StrLngt;

if(Str1.length()<=Str2.length())
StrLngt=Str1.length();
else
StrLngt=Str2.length();
```

```
for(int j=0;j<StrLngt;j++){
if(Str1.charAt(j)!=Str2.charAt(j)){
StrLngt=Str1.length()+Str2.length()-2*j;
break;
}
}
System.out.println("the string subis:   "+StrLngt);
return StrLngt;

}

}


/*
* To change this template, choose Tools | Templates
* and open the template in the editor.
*/

package puzzlesolver;
import java.util.*;
/**
*BFS Functions for generating nodes  and checking duplication nodes
* @author kose
*/
public class BFSMoves {

public Node  Left(Node N, int Empty){
Node NewNode = new Node();
Heuristic H= new Heuristic();
System.arraycopy(N.board, 0, NewNode.board, 0, N.board.length);
NewNode.board[Empty] = N.board[Empty-1] ;
NewNode.board[Empty-1]=0;
//NewNode.MoveWay=0;
// NewNode.MoveIndex=N.MoveIndex*4;

NewNode.depth=N.depth+1;
return NewNode;
}

public Node Right(Node N,int Empty){

Node newConf=new Node();
System.arraycopy(N.board, 0, newConf.board, 0, 16);
newConf.board[Empty] = N.board[Empty+1] ;
newConf.board[Empty+1]=0;
```

```
newConf.depth=N.depth+1;
newConf.Parent=N;
return newConf;
}

public Node Down(Node N, int Empty){
// Move the Empty Tile to down.
Node NewNode=new Node();
System.arraycopy(N.board, 0, NewNode.board, 0, 16);
NewNode.board[Empty] = N.board[Empty+4] ;
NewNode.board[Empty+4]=0;
NewNode.depth=N.depth+1;
NewNode.Parent=N;
return NewNode;
}

public Node Up(Node N,int Empty){
// Move the Empty Tile to up.
Node NewNode=new Node();
System.arraycopy(N.board, 0, NewNode.board, 0, 16);
NewNode.board[Empty] = N.board[Empty-4];
NewNode.board[Empty-4]=0;
// NewNode.MoveWay=3;
// NewNode.MoveIndex=N.MoveIndex*4+3;
NewNode.depth=N.depth+1;
NewNode.Parent=N;
return NewNode;
}



//
/*
* Check Duplication For breath Firt Search
*/

public boolean CheckDuplicationBFS(Node dup,LinkedList <Node>Lst){

if(Lst.isEmpty())
return false;
boolean flag=true;
ListIterator <Node>Litr = Lst.listIterator();
while(Litr.hasNext()) {
flag=true;
Node N = Litr.next();
search:
```

```
for(int i=0;i<dup.board.length;i++){
if(dup.board[i]!=N.board[i]){
flag=false;
break search;
}
}
if(flag)
return flag;
}
return flag;
}
}


/*
* To change this template, choose Tools | Templates
* and open the template in the editor.
*/

package puzzlesolver;
import java.lang.Math.*;

/**
* Manhattan Distance Heuristic
* @author kose
*/

public class Heuristic {
public int H(int Conf[], int Conf2[]){

int Dist=0;
int PairDist=0;
int x=0;
// int[] a = new int[2];
for (int i = 0; i<Conf.length; i++){
if(Conf[i]!=0){
search:
for(int j=0;j<Conf2.length;j++)
if(Conf[i]==Conf2[j]) {
x=ManhattanDist(i,j);
Dist=Dist+x;

break;
}
}

}
```

```
// PairDist=PairDist+PairDistance(Conf,Conf2);

return (Dist+PairDist);

}

public int PairDistance(int S[], int G[]){
int PairDist=0;
int MD=0;
for(int s=0;s<S.length;s++){
if(S[s]!=0){
for(int g=0;g<G.length;g++){
if(G[g]!=0){
if(S[s]==G[g]){
MD=ManhattanDist(s,g);
if(s%4==g%4 & s!=g){ // if two tiles are in the same column
if (g<s){
for(int g1=g;g1<s;g1=g1+4){

for(int n=g%4;n<15;n=n+4){
if(S[g1]==G[n])
PairDist=PairDist+2;
}
}

}else{
for(int s1=s-4;s1>=g;s1=s1-4){
for(int n=s%4;n<15;n=n+4){
if(S[s1]==G[n]){
PairDist=PairDist+2;
}
}
}
}
}

else if(Math.abs(s-g)==MD &(s!=g)){
if (g<s){
for(int g1=g+1;g1<s;g1=g1+1){
for(int n=0;n<4;n=n+1){
int x=(s/4)*4;
if(S[g1]==G[x+n]){
PairDist=PairDist+2;
}
}
}
```

```
}else{

for(int s1=s+1;s1<=g;s1=s1+1){
for(int n=0;n<4;n=n+1){
int x=(s/4)*4;
if(S[s1]==G[x+n])
PairDist=PairDist+2;
}

}
}

} //

}// end of the second iff
}

}// end of second for
} // end of the first if

} //end of first for
return PairDist;
}
public int ManhattanDist(int Tile, int j) {
int Mod= Math.abs((j%4- (Tile)%4))+ Math.abs((j / 4- (Tile)/4));
return Math.abs(Mod);
}

}
```

15. References

[1] Berlekamp, E. R., Conway, J. H., and Guy, R. K. (2000). **Winning Ways for Your Mathematical Plays**. Academic Press, Vol. I, II.

[2] Bolac, L. and Cytowski, J. (1992). **Search Methods for Artificial Intelligence**. *Academic Press.*

[3] Chameaux, D. and Sint, S. (1977). An Improvement on Bidirectional Heuristic Search Algorithm. *JACM,* Vol. 24 (2).

[4] Chameaux, D. (1983). Bidirectional Heuristic Search Again. *JACM.* Vol. 30 (1) p.p. 22-32.

[5] Culberson, J. C. and Schaeffer, J. (1996). Search with Pattern Databases. *In Springer Berlin /Heidelberg.* pp. 402-416.

[6] Culberson, J. C. and Schaeffer, J. (1998). Pattern Databases. *Computational Intelligence.* Vol. 14 (3), pp. 318-334.

[7] Dantzig, G. B. (1960). On the Shortest Route through a Network. *Management Science.* Vol. 6 (2), pp. 187-190.

[8] Dillenburg, J. F. and Nelson, P. C. (1994). Perimeter Search. *Artificial Intelligence.* Vol. 65, pp. 165-178.

[9] Felner, A., Korf, R. E. and Hanan, S. (2004). Additive Pattern Database Heuristics. *Journal of Artificial Intelligence Research.* pp. Vol.22, pp. 279-318.

[10] Felner, A., Moldenhauer, C., Sturtevat, N. and Schaeffer, J. (2010). Single-Frontier Bidirectional Search. *AAAI.* Atlanta, Georgia, USA.

[11] Fu, L., Sun, D. and Rilett, R. L. (2006). Heuristic Shortest Path Algorithms for Transportation Applications: State of the Art. *Computer and Operations Research.* pp.Vol 33 (11), pp. 3324-3343.

[12] Ishida, T. (1995). Two is not Always Better than One: Experiences in Real Time Bidirectional Search. *Proceedings of the International Conference on Multi-Agent Systems.* Pp. 185-192

[13] Kaindl, H. and Kainz, G. (1997). Bidirectional Heuristic Search Reconsidered. *Journal of Artificial Intelligence*. Vol. 7, pp. 283-317.

[14] Kalos, M. H. and Whitlock, P. (2008). **Monte Carlo Methods 2nd ed.** New Yok, *Wiley-VCH.*

[15] Kirkpatrick, S., Gelett, C.D., and Vechi, M. P. (1983). Optimization by Simulated Annealing. *Science*. Vol. 22, pp. 621-630.

[16] Kopec, D. Marsland, T. A., and Cox, J. (2004). Search.*The Computer Science and Engineering Handbook, (2nd ed., Ed. A. Tucker), CRC Press, Boca Raton, FL.* Chapter 26, pp. 1-26.

[17] Korf, R., E. and Felner, A. (2002). Disjoint Pattern Database Heuristics. *Artificial Intelligence.* Vol.134, pp. 9-22.

[18] Korf, R.E. (1993). Linear-space Best First Search. *Artificial intelligence*. Vol. 62 (2) , pp.41-78.

[19] Korf, R. E. (1999). Divide- and-Conquer Bidirectional Search: First results. *In Proc. of 16th International Joint Conference on Artificial intelligence (IJCAI-99*). Vol. 2, pp. 1184-1189.

[20] Korf, R. E. (1985). Depth-First Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence.* Vol. 27 (1), pp. 97-109.

[21] Kwa, J. (1989). BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm. *Artificial Intelligence.* Vol. 38(2), pp. 95-109.

[22] Land, A. H. and Doig, A. G. (1960). An Automatic Method of Solving Discrete Programming Problems. *Econometrica*. Vol. 28, pp. 497-520.

[23] Landin, P. J. (1966). The Next 700 Programming Languages. *"Communication of the AMC".* Vol. 9 (1).

[24] Lee, C. Y. (1961). An Algorithm for path connection and its applications. *IRE Transactions on Electronic Computers.* Vol. EC – 10 (3), pp. 346-365.

[25] Luger, G. F. and Stubblefield W. A. (2009). **AI Algorithms, Data Structures, and Idioms in Prolog, List, and Java**. Boston, MA. Pearson Education, Inc.

[26] Luger, G. F. (2002). **Artificial Intelligence Structures and Strategies for Complex Problem Solving, 4th ed**. Reading, MA. Addison Wesley.

[27] Manzini, G. (1996). Perimeter search in Restricted Memory. *Computers Math. Applica.* Vol.32 (7), pp.37-45.

[28] Manzini, G. (1995). BIDA* : An Improved Perimeter Search Algorithm. *Artificial Intelligence.* Vol. 77, pp. 347-360.

[29] Michalewicz, Z. and D. B. Fogel. (2000). **How to Solve It: Modern Heuristic.** New York, Springer – Verlag: Berlin Heidelberg.

[30] Moore, F. E. (1959). The shortest path through a maze. *In Proceeding of the International Symposium on the Theory of Switching*. Harvard University Press. pp. 285-292.

[31] Nicholson, T. A. (1966). Finding the Shortest Route Between Two Points in a Network. *Computer Journal.* Vol. 9 (3), pp. 275-280.

[32] Pijls, W. and Post, H. (2009). A New Bidirectional Search Algorithm with Shortest Post- Processing. *European Journal of Operational Research.* Vol.198 (2), pp.363-369.

[33] Pearl, J. (1984). **Heuristics, intelligent search strategies for computer problem solving**. Reading, MA. Addison – Wesler.

[34] Pohl, I. The Bi-directional Search. (1971). *In Machine Intelligence.* Vol. 6, pp. 127-140.

[35] Russell, S. J. and Norvig, P. (2003). **Artificial Intelligence: A Modern Approach, 2nd ed.** Prentice Hall.

[36] Schaeffer, j., Bjornsson, Y., Burch, N., Kishimoto, A., Muller, M., Lake, R., Lu P., and Sutphen, S. (2007). Solving Checkers. *Science Daily*, July 20.

[37] Schaeffer, J. (2007). Checkers is Solved. *Science AAAS.* 1518, 317.

[38] Tucker, A. B. and Noonan, R. E. (2007). **Programming Languages Principles and Paradigms 2nd ed.** Mcgraw Hill.

[39] Webber, A. B (2002). **Modern Programming Languages**. Franklin, Beedle And Associates Incorporated.

[40] Zahavi, U., Felner, A., Holte, R., and Schaeffer ,J. (2006). Dual Search in Permutation State Spaces. *AAAI. Boston, MA, USA.*

[41] Zahavi U., Felner A., Holte R., and Schaeffer J. (2008). Duality in the Permutation Spaces and the Dual Search Algorithm. *Artificial Intelligence.* Vol. 172 (4-5), pp. 514-540.

[42] Zhang, W. (1999). **State Space Search- Algorithms, Complexity, Extensions and Applications**. New York, Springer – Verlag Berlin Heidelberg.

[43] Zhou, R. and Hansen, E. A. (2004). Space-Efficient Memory-Based Heuristics. *AAAI.* San Jose, California.

[44] Zhou, R. and Hansen, E. A. (2004). Breadth-First Heuristic Search. "*AAAI*. San Jose, California.