

CS351 Spring 2004, Project 2

PuzzleMuncher

MondoSoft, Inc.

March 24, 2004

STATUS Specification and requirements document

VERSION 1.1

DATE Mar 24, 2004

0 Changelog

Version 1.1 Fixed bug in grammar description. Mar 24, 2004.

Version 1.0 Initial release. Feb 23, 2004.

1 Summary

Once again, your fantastically insightful managers at MondoSoft have identified a stellar business opportunity. This time, they have observed the growing popularity of puzzle games, and the growing popularity of laziness and decided to combine the two into the ultimate puzzle assistive technology for the ultimately lazy. The new product will be `PuzzleMuncher`, a program for automatically solving puzzle games. `PuzzleMuncher` will input a description of a puzzle initial configuration and goal, find the optimal path between the initial state and goal, and output the sequence of moves that solves the puzzle. To increase broad-spectrum marketability of the `PuzzleMuncher` system, it must be capable of solving multiple different kinds of puzzles, and support a user API for extending the system to additional puzzle types.

2 Definitions

ACTION The process by which some `STATE` is transformed into one of its `SUCCESSOR STATES`. The application of an `ACTION` to a particular `STATE` **MUST** yield a single `SUCCESSOR STATE`.

ADMISSIBLE A `HEURISTIC` is `ADMISSIBLE` if and only if it underestimates the distance from the current state to the `GOAL STATE`. That is, $h(s)$ is `ADMISSIBLE` iff $h(s) \leq g(\text{goal}) - g(s) \forall s$.

CHILD STATE See, SUCCESSOR STATE.

CLOSED LIST Set of all STATES that have been fully examined (i.e., had their SUCCESSORS generated).

CONSISTENT See, MONOTONIC.

GOAL STATE The terminal STATE of a PUZZLE. When the shortest sequence of MOVES between the START STATE and some GOAL STATE is found, the PUZZLE has been solved.

HEURISTIC A function, denoted h , that estimates the remaining cost from the current state to the goal state. Important special cases are ADMISSIBLE and MONOTONIC HEURISTICS.

MAY A requirement that the product can choose to implement if desired. Can also indicate a choice among acceptable alternatives (e.g., “The program MAY do x, y, or z.” indicates that the choice of behavior x, y, or z is up to the designer.)

MONOTONIC Also known as CONSISTENT. A heuristic is monotonic if and only if it underestimates the cost from every state to each of their SUCCESSOR STATES. That is, $h()$ is MONOTONIC iff $h(s) \leq c(s, a, s') + h(s')$ for any STATE s having SUCCESSOR STATE s' under ACTION a .

MOVE An operation applied to a STATE to produce another STATE.

MUST A requirement that the product must implement for full credit.

MUST NOT A behavior or assumption that must not be violated. Violating a MUST NOT restriction will result in a penalty on the assignment.

NODE See, STATE.

OPEN LIST Set of STATES that have been generated (as children of some other node), but not yet examined.

PARENT STATE See, PREDECESSOR STATE.

PREDECESSOR STATE Also called a PARENT STATE. The STATE to which a MOVE is applied to generate a SUCCESSOR STATE.

PUZZLE A single-player game with a defined START STATE and one or more GOAL STATES. From any STATE one or more MOVES are possible, which transform the STATE into a SUCCESSOR STATE. The object is to find the shortest sequence of MOVES which transform the START STATE to the GOAL STATE.

RECOVERABLE ERROR An error condition that the software can ignore, correct, or otherwise recover from. The program MUST produce a warning message and then cleanly continue with no corruption or loss of valid data.

START STATE The initial configuration of a PUZZLE.

STATE A possible configuration of a PUZZLE. Also known as a NODE, when conceived of as an element of the search graph.

SYNTAX ERROR A discrepancy between the input file and the grammar given in Section 3.2.

SUCCESSOR STATE Also called CHILD STATE. A STATE generated by the application of a MOVE to a PREDECESSOR (or PARENT) STATE.

UNRECOVERABLE ERROR An error condition from which recovery is impossible. The program MUST produce an error message describing the condition and then cleanly halt.

3 Requirements

This section describes the elements that MUST be developed as part of this project. The designer MAY also choose to implement additional Java source files, programs, and/or shell scripts in support of the following items. This section only describes the general performance requirements for each element; for specific deliverable requirements, please refer to Section 5.

3.1 Puzzles

The `PuzzleMuncher` software will consist of one required program, `PuzzleMuncher`, which will be capable of solving at least the three different PUZZLES specified in this section. In addition, the `PuzzleMuncher` system MUST support the specified `PuzState` interface so that additional puzzles can be seamlessly incorporated. The designer MAY also provide additional PUZZLE implementations with the final project.

3.1.1 The Missionaries and Cannibals Puzzle

In this puzzle, there are m missionaries and c cannibals on the one bank of a river, together with a boat that can carry p passengers. Initially, $m \leq c$. The object is to reach a state in which all missionaries and cannibals are on the West bank of the river. The difficulty is that if the missionaries ever outnumber the cannibals on either side of the river, that group of missionaries eats their unfortunate cannibal associates¹. To successfully solve the puzzle, no cannibals may be devoured.

A MOVE consists of selecting a group of one or more passengers to load in the boat, rowing the boat across the river, and unloading the boat on the other side. Note that the boat cannot cross the river with no passengers and the boat cannot carry more passengers than its capacity. The river is full of hungry piranha and anybody attempting to swim will be devoured. The boat MUST be fully unloaded on the far side of the river, at which point the assessment of conversions is performed.

For the purposes of output, a MOVE is represented by:

MOVE i : FERRY m_c Cannibals and m_m Missionaries FROM s_i TO s_f

where m_c and m_m denote the number of cannibals and missionaries relocated during this move, respectively, and s_i and s_f denote the initial and final sides of the river. s_i and $s_f \in \text{East, West}$ and $s_i \neq s_f$.

¹They're missionaries of the church of Cthulhu.

	1	2
3	4	5
6	7	8

(a)

3	1	2
	4	5
6	7	8

(b)

Figure 1: Example of two $n^k - 1$ puzzle ($n = 3, k = 2$) STATES, joined by the move $[2, 1] \rightarrow [1, 1]$.

The output of a state is the same as the input format, specified by the rule **MCSTATE** in Section 3.2.

The null move (moving no people) is disallowed.

3.1.2 The Shortest Paths Puzzle

In this “puzzle”, the system is presented with a description of a map, giving a list of cities and the distances between various pairs of cities. The initial STATE is a starting city and the final STATE is a goal city. The object is to find the shortest path between the two cities. A MOVE consists of traveling from the current city to one of its adjacent cities.

For the purposes of output, a MOVE is represented by:

MOVE i : TRAVEL FROM c_j TO c_k

where $i \in 0, 1, \dots$ represents the number of the current move and c_j and c_k are the names of the origin and destination cities, respectively.

The output of a state is specified by the BNF rule

SOUTPUT := "CurrentCity" "=" **CITYNAME**

where **CITYNAME** is given by the rule in the input grammar in Section 3.2.

The null move (changing nothing) is disallowed.

Note that the distances between cities is *not* required to be symmetric or to obey the triangle inequality!

3.1.3 The $n^k - 1$ Puzzle

This PUZZLE is played on a hypercubical grid of $n \times n \times n \cdots \times n$ elements. Each grid hypercube can be occupied by a single numbered tile, or may be empty. A single move consists of sliding a tile from its current position into an adjacent empty grid location. Figure 1 illustrates two STATES of this PUZZLE. Figure 1 (b) arises from moving the tile “3” in configuration (a) into the empty grid location above it. A tile may only be moved along a single dimension at a time; “diagonal” moves are not allowed. Thus, in Figure 1 (a), it would be illegal to move the tile “4” into the empty grid location.

The object of this PUZZLE is to take a START STATE, given by an initial tile configuration, and a GOAL STATE, given by a desired tile configuration, and produce the shortest sequence of MOVES that produce the GOAL from the START.

The PuzzleMuncher program MAY assume that all instances of the $n^k - 1$ puzzle have only a single blank tile, though they MAY choose to support multiple blank tiles. Furthermore, it MAY assume that every numbered tile is unique. If either of these conditions is violated, the PuzzleMuncher system MAY treat this as a RECOVERABLE or UNRECOVERABLE error,

or it MAY silently handle it. If the `PuzzleMuncher` chooses to handle this case, it MUST do so correctly.

For the purposes of output, every MOVE is given by the expression:

MOVE i : $[p_1, p_2, \dots, p_k] \rightarrow [p'_1, p'_2, \dots, p'_k]$

where $i \in 0, 1, \dots$ denotes the current move, the left hand expression gives the location of the tile to be moved, and the right hand expression denotes the location into which it is moved. The values p_i give the index along each dimension of the puzzle. Note that because of restriction that moves may only be along a single dimension, $p'_j = p_j$ for all but one j .

The output of a board state is the same as the input format, specified by the rule **NKPUZSTATE** in Section 3.2.

The null move (changing nothing on the board) is disallowed.

3.2 Input File Grammar

The input to the `PuzzleMuncher` consists of a single description file given as the sole (normal) command-line argument to the program. The programmer MAY also provide additional command-line arguments (optionally with the `gnu.getopt` package) at her or his discretion. Any such additional options MUST be documented in the user documentation for `PuzzleMuncher`.

The input file is given by the following BNF grammar. The program MAY assume that all input data is standard, 7-bit ASCII; deviations from this MAY be treated as RECOVERABLE or UNRECOVERABLE ERRORS or silently ignored. The start symbol of this grammar is **FILE**.

```

FILE                := ( CONTROL | PUZZLEDEF ) *
CONTROL             := ( OUTFILE |
                           LOGFILE |
                           ERRFILE |
                           RESULTS |
                           STATS |
                           SEARCH-CTRL |
                           "Run" |
                           "Reset" )
OUTFILE             := "OutFile" "=" FILESPEC
LOGFILE             := "LogFile" "=" FILESPEC
ERRFILE             := "ErrFile" "=" FILESPEC
FILESPEC            := "\" " PATH? DIR-OR-FILENAME "\" "
PATH                := "/"? ( DIR-OR-FILENAME "/" )+
DIR-OR-FILENAME    := [a-zA-Z0-9._-]+
RESULTS             := ( "SolnPathLen" | "StatePath" | "MoveSeq" )
STATS               := ( "NodesOpened" |
                           "OpenListMaxLen" |
                           "NodesClosed" |
                           "NumReopened" |
                           "OpenClosedRatio" )
SEARCH-CTRL        := ( OPENLIST-BOUND | TOTALNODES-BOUND | TIME-BOUND )
OPENLIST-BOUND     := "OpenListBound" "=" POS-INTEGER

```

```

TOTALNODES-BOUND := "TotalNodesBound" "=" POS-INTEGER
TIME-BOUND       := "TimeBound"   "=" POS-INTEGER
PUZZLEDEF        := "Puzzle"
                  ( MCPUZZLE |
                    SPPUZZLE |
                    N2KPUZZLE )
MCPUZZLE         := "MissionariesAndCannibals" "(" HNAME ")"
                  "=" "{"
                  "InitialState" "=" MCSTATE
                  "GoalState"   "=" MCSTATE
                  "BoatCapacity" "=" POS-INTEGER
                  "}"
MCSTATE          := "{"
                  "WEST" "BANK" ":" BANKSPEC
                  "EAST" "BANK" ":" BANKSPEC
                  "BOAT" "is" "on" ( "EAST" | "WEST" ) "BANK"
                  "}"
BANKSPEC         := NON-NEG-INTEGER "Cannibals" "and"
                  NON-NEG-INTEGER "Missionaries"
SPPUZZLE         := "ShortestPaths" "(" HNAME ")"
                  "=" "{"
                  "Cities"      "=" CITYLIST
                  "Distances"   "=" DISTLIST
                  "StartCity"    "=" CITYNAME
                  "GoalCity"     "=" CITYNAME
                  "}"
CITYLIST         := "(" ( CITYNAME ( "," CITYNAME )* )? ")"
DISTLIST        := "(" ( DISTPAIR ( "," DISTPAIR )* )? ")"
DISTPAIR        := CITYNAME "->" CITYNAME "=" NON-NEG-INTEGER
CITYNAME        := [A-Z][a-zA-Z]+
N2KPUZZLE       := "NToTheKPuzzle" "(" HNAME ")"
                  "=" "{"
                  "StartState"  "=" NKPUZSTATE
                  "GoalState"   "=" NKPUZSTATE
                  "}"
NKPUZSTATE      := "[" ( NUMLIST |
                  NKPUZSTATE ( "," NKPUZSTATE )* )
                  "]"
NUMLIST         := NON-NEG-INTEGER ( "," NON-NEG-INTEGER )*
HNAME          := [a-zA-Z]+
POS-INTEGER     := [1-9][0-9]+
NON-NEG-INTEGER := [0-9]+

```

This grammar is case sensitive so that, for example, "OpenListBound" does not match

"openlistbound". Tokens are separated by whitespace (as defined by `java.lang.Character.isWhitespace()`), though whitespace is otherwise ignored by the grammar.

A SYNTAX ERROR in the input data file MAY be treated as a RECOVERABLE or UNRECOVERABLE ERROR.

The designer MAY implement additional syntax if desired, for example, to support additional PUZZLE types, or additional **RESULTS**, **STATS**, or **SEARCH-CTRL** mechanisms. All such additional syntax (and the corresponding semantic extensions) MUST be documented in the appropriate API and user documentation.

3.2.1 Semantics

The commands, given syntactically in the grammar, have the following interpretations:

"ErrFile" Set the destination file for any error message outputs to the specified filename. If the file cannot be opened, this can be considered to be a RECOVERABLE or UNRECOVERABLE ERROR. `PuzzleMuncher` MUST be able to correctly handle the case in which this file name is the same as "OutFile" or "LogFile". If an `ErrFile` is specified, and then a different `ErrFile` is given later, the first file is flushed and closed and then the new file is opened. If the new file is identical to the old, the new data should be *appended* to the old data — no data should be lost by this operation. **Default value:** Standard error.

"LogFile" Set the destination file for the performance statistics output to the specified filename. If the file cannot be opened, this can be considered to be a RECOVERABLE or UNRECOVERABLE ERROR. `PuzzleMuncher` MUST be able to correctly handle the case in which this file name is the same as "OutFile" or "ErrFile". If a `LogFile` is specified, and then a different `LogFile` is given later, the first file is flushed and closed and then the new file is opened. If the new file is identical to the old, the new data should be *appended* to the old data — no data should be lost by this operation. **Default value:** Standard output.

"OutFile" Set the destination file for puzzle results output to the specified filename. If the file cannot be opened, this can be considered to be a RECOVERABLE or UNRECOVERABLE ERROR. `PuzzleMuncher` MUST be able to correctly handle the case in which this file name is the same as "LogFile" or "ErrFile". If an `OutFile` is specified, and then a different `OutFile` is given later, the first file is flushed and closed and then the new file is opened. If the new file is identical to the old, the new data should be *appended* to the old data — no data should be lost by this operation. **Default value:** Standard output.

"Run" Perform a search on the most recently defined PUZZLE and provide any output required by current **RESULTS** and **STATS**. If any of the bounds specified by **SEARCH-CTRL** are hit, the search MUST terminate and report no solution found, along with the reason for search termination. In this case, the **RESULTS** need not be reported (other than no solution found), but any requested **STATS** MUST be.

"Reset" Reset all **RESULTS**, **STATS**, and **SEARCH-CTRL** settings to their default values. Note that this does not reset any of the output files; those remain until explicitly overridden or until the program terminates.

- "SolnPathLen"** Report the number of MOVES in the solution path (i.e., the number of MOVES required to transform the START STATE to the GOAL STATE).
- "StatePath"** Report the complete sequence of STATES encountered on the solution path, including the START and GOAL STATES. The output format for each state is specified under each puzzle in Section 3.1.
- "MoveSeq"** Report the actual MOVES taken along the solution path. The output format for each move is specified under each puzzle in Section 3.1. Note that this is distinct from **SolnPathLen** which only reports the *number* of moves along the solution path.
- "NodesOpened"** Report the total number of nodes opened during the search.
- "OpenListMaxLen"** Report the maximum number of nodes on the OPEN LIST at any point during the search.
- "NodesClosed"** Report the total number of nodes on the CLOSED LIST at the end of the search.
- "NumReopened"** Report the number of nodes moved from the CLOSED LIST to the OPEN LIST during the search.
- "OpenClosedRatio"** Report the maximum and minimum ratios of the size of the OPEN LIST to the size of the CLOSED LIST during any point in the search. (For the purposes of this report, you can consider $0/0 = 1$ and $x/0 = x$.)
- "OpenListBound"** Specifies the maximum number of elements that may be in the OPEN LIST. If this bound is exceeded, terminate the search. **Default value:** `java.lang.Integer.MAX_VALUE`.
- "TotalNodesBound"** Specifies the maximum number of nodes that can be in the search at one time (total of OPEN LIST and CLOSED LIST). If this bound is exceeded, terminate the search. **Default value:** `java.lang.Integer.MAX_VALUE`.
- "TimeBound"** Maximum running time of the search, given in milliseconds (wall clock time, as reported by `java.lang.System.currentTimeMillis()`). If this bound is exceeded, terminate the search. **Default value:** `java.lang.Long.MAX_VALUE`.
- HNAME** This argument to a puzzle description specifies the heuristic that will be used to solve this puzzle during the current search iteration.

By default, no **RESULTS** or **STATS** are generated — all outputs must be explicitly specified.

All **RESULTS**, **STATS**, and **SEARCH-CTRL** settings are durable — after being set, they remain in that state across multiple searches, until another value is assigned, a "Reset" is issued, or the search program terminates.

When the end of the input specification file is reached and all commands in the file have been completed, the program terminates. At termination, the program **MUST** flush and close all open files.

3.3 The PuzState, Monotonic, and Parseable Interfaces

The `PuzzleMuncher` system **MUST** be capable of accepting any puzzle game that obeys the `PuzState` interface, as given at

<http://www.cs.unm.edu/~terran/classes/cs351-s04/project-2/PuzState.java>

This interface specifies methods for the distance from start, distance to goal, and heuristic functions.

In addition, `PuzzleMuncher` must also distinguish between monotonic and non-monotonic heuristics and apply the most efficient available algorithm, depending on the case at hand. To handle this, `PuzzleMuncher` **MUST** recognize the `Monotonic` marker interface and employ the appropriate algorithm. Any puzzle state object which implements the `Monotonic` interface **MUST** guarantee monotonicity of its heuristic. Thus, the A* engine **MAY** assume that the heuristic provide by such an object is, in fact, always monotonic. If the A* engine detects a deviation from this property, it **MAY** treat this condition as a `RECOVERABLE` or `UNRECOVERABLE` error. The `Monotonic` interface is given at:

<http://www.cs.unm.edu/~terran/classes/cs351-s04/project-2/Monotonic.java>

Finally, to support parsing and printing of objects, a `Parseable` interface is specified which provides a `loadYourself` method to support parsing and `printYourself` and `printLastMove` methods to support output. The `Parseable` interface is available at:

<http://www.cs.unm.edu/~terran/classes/cs351-s04/project-2/Parseable.java>

4 Quantitative Requirements

This section describes the performance and IP requirements for the `PuzzleMuncher` software.

- All programs **MUST NOT** crash, core dump, dump a stack trace, or throw an exception on any input.
- In the case of a `RECOVERABLE ERROR`, a program **MUST** issue a warning statement and continue processing. The program **MAY** choose to issue the warning statement to standard error or to a log file. If the warning is issued to a log file, the log file name and location **MUST** be a user-specifiable parameter to the program.
- In the case of an `UNRECOVERABLE ERROR`, a program **MUST** issue an error statement and terminate with a non-zero error condition. The program **MAY** use different exit codes to indicate different error conditions, but such codes **MUST** be documented in the user manual. The error message **MUST** be logged to the same destination that warning messages (from `RECOVERABLE ERRORS`) are.
- In the case of any termination (whether normal, because a `SEARCH-CTRL` bound was met, or because of an `UNRECOVERABLE ERROR`), the program must cleanly flush and close all open files. No other data may be lost or destroyed.

- All normal output **MUST** be sent to the file(s) specified by the **OUTFILE**, **ERRFILE**, and **LOGFILE** commands. No other output may be generated, by default.
- The programs **MAY** provide additional output for debugging purposes, *but* such output must be *disabled by default*. Any program **MAY** provide a command-line switch to enable debugging support when desired.
- The search routine **MUST** recognize **MONOTONIC** heuristics and employ the most efficient possible search algorithm for them.
- The **OPEN LIST** data structure **MUST** be able to support the operations `insert`, `remove-first`, `remove`, and `reinsert` in $O(\log n)$ time for a list containing n nodes. For extra credit, it **MAY** support one or more of these operations in $O(1)$ or amortized $O(1)$ time. To receive the extra credit, this feature **MUST** be documented in the user and API documentation for the search engine, the reasons for the improved performance **MUST** be described, and some performance documentation **MUST** be provided demonstrating the claimed time bounds.
- For full credit, the `PuzzleMuncher` program **MUST NOT** use or reference any of the built-in lexical analyzers provided by the JDK (i.e., `java.util.StreamTokenizer`, `java.util.StringTokenizer`, or `java.util.regex.*`).
- The `PuzzleMuncher` program **MAY** use the `gnu.getopt.Getopt` and `gnu.getopt.LongOpt` classes to assist in handling command-line options, if the programmer feels that it is useful to do so.
- The `PuzzleMuncher` program **MAY** assume that all valid input is standard ASCII text in the range `(char)0–(char)127`, inclusive. If a program encounters a character outside this range, it **MAY** treat it as a **RECOVERABLE** or **UNRECOVERABLE ERROR** or silently ignore it. If such characters are treated as **RECOVERABLE** or ignored, they **MUST NOT** disrupt the otherwise normal functioning of the program.
- The program **MUST NOT** assume that the input is syntactically correct. If a syntax error occurs, the program **MAY** treat it as a **RECOVERABLE** or **UNRECOVERABLE ERROR** or silently ignore it. In all cases, the program **MUST** handle the error gracefully, reporting it only to the **LOGFILE** destination and terminating cleanly if necessary. It **MUST NOT** crash, core dump, or generate a stack trace. Any files that are open at termination **MUST** be cleanly flushed and closed.
- The programmer **MAY** ask permission of the instructor or the TA to use any classes outside the JDK that have not already been mentioned. The final programs **MUST NOT** use any class outside the JDK that have not been explicitly allowed.
- All user documentation **MUST** be grammatically correct and include correct spelling and usage.
- The programmer **MUST** document any areas in which her or his software suite does not meet this specification. **WARNING!** The grade penalty will be higher if the instructors discover an undocumented program shortcoming or bug than if it is documented up front.

5 Deliverables

This section describes the content to be delivered at each stage of the project (two milestones and a final rollout). For the deadlines of these stages, please refer to Section 6.

5.1 Milestone 1: Analysis

The first deliverable consists of an analysis and design phase. The designer is to prove correctness of the search algorithm in question and to give a preliminary design for the necessary object(s). This entire milestone will take the form of a single written document answering each of the following questions:

1. Prove that under an admissible heuristic, the A* algorithm is guaranteed to find the shortest path to the goal state. Specifically, when the goal state is opened, the path by which it was discovered is an optimal path.
2. Prove that under a monotonic heuristic, the A* algorithm is guaranteed to find the shortest path to *every* state (including the goal state). Specifically, the first time any state is opened, the path by which it was discovered is an optimal path to that state.
3. Describe and provide pseudo-code for an algorithm+data structure that will implement the A* search for a general, admissible puzzle. This design need not be at the full resolution of Java code (i.e., it doesn't have to provide full type information, full bounds and error checks, etc.), but it should be at a much lower level than the general A* description that will be discussed in class or that is found in most AI textbooks. Specifically, it must provide a description of and pseudocode for the underlying data structures necessary to implement the OPEN and CLOSED LISTs, as well as the Java method signatures for all public interfaces with the OPEN/CLOSED list and the entire search engine object.

For example, simply stating that "The OPEN LIST will be implemented in terms of a queue" is insufficient. You must also specify how that queue is implemented, what methods it supports, pseudocode for those methods, etc.

4. Provide an analysis of the running time and space overhead for each of the methods specified in Question 3. A full, formal proof is not always necessary, but a reasonably detailed and convincing analysis is.
5. Repeat Questions 3 and 4 for the special case of a puzzle with a monotonic heuristic. You need not re-specify any components that are unchanged in this version, but you must specify all changes and give pseudocode for any algorithms that are modified. If improved time/space bounds can be obtained for this version, describe how and why as well as which data structure(s) take advantage of those changes.
6. Give at least two nontrivial, ADMISSIBLE HEURISTICS for each of the PUZZLES described in Section 3.1. For each HEURISTIC, specify whether it is simply admissible or whether it is also MONOTONIC. Can you construct a HEURISTIC that is ADMISSIBLE but *not* MONOTONIC?

This milestone MAY be handwritten or MAY be typed/word processed, but it MUST be neat and legible in all cases. Pay special attention to logical flow, support of conclusions, and clarity of writing.

Either paper or electronic submission is acceptable for this milestone.

NOTE: Although this milestone only describes design, it is expected that the designer/implementer will also be working on code aspects of the project in parallel with accomplishing this milestone.

5.2 Milestone 2: The A* Engine

In this milestone, the designer/implementer will deliver a functional search engine object capable of solving puzzles with either a general admissible or a monotonic heuristic. This milestone does *not* specifically include implementations of the PUZZLEs described in Section 3.1, though it may be necessary to develop one or more PUZZLE objects in order to test the system. The instructors will test the search engine object with an arbitrary PUZZLE object obeying the interface specifications given in Section 3.3. This milestone also does not include support for the input file format specified in Section 3.2, though some sort of unit testing is required and limited parsing ability MAY be supported for that purpose. Specific materials to be handed in will be:

Java code The class file(s) for all objects necessary to compile and instantiate the search engine object, including any supporting code necessary to compile, load, and use the search engine module. Note that this may include one or more PUZZLE objects in order to test the functionality of the engine.

API documentation The handin MUST also include the full, compiled JavaDoc documentation for the search engine object(s) implementation. This documentation MUST include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed in the code. This documentation hierarchy MUST be included in a sub-directory named `documentation/` within the submission tarball package.

Unit test documentation The handing MUST include a document named `PERFORMANCE.extension` that describes unit tests on this module and demonstrates correct functionality.

Test cases If any external test data is necessary for the unit tests of this object, that data MUST be submitted in the subdirectory `tests/` within the tarball.

CVS log file(s) For each Java source file, the submission tarball MUST include a corresponding `.log` file including the CVS log for that sourcecode.

At the programmer's option, this submission MAY also include:

BUGS.TXT This file documents any known outstanding bugs, missing features, performance problems, or failures to meet specifications of your submission. Note that the penalty for such problems will be smaller if they're fully documented here than if the instructors discover them independently.

The submission directory MUST be named `lastname_p2m2` and the submission tarball MUST be named `lastname_p2m2.tar.gz`.

5.3 Rollout: The PuzzleMuncher System

The final stage of this project is delivery of the complete `PuzzleMuncher` system, including ability to parse the complete description file specified in Section 3.2 and support for all `PUZZLES` specified in Section 3.1, as well as performance tests demonstrating the relative performance of different heuristics and the scalability of the search algorithm with respect to the size and difficulty of the puzzles. This submission **MUST** include support for at least the heuristics described in Milestone 1, as well as any additional heuristics the designer wishes to provide.

The deliverables for this stage are:

`PuzzleMuncher.java` The primary object of the `PuzzleMuncher` program suite.

Other Java source files Any other supporting code necessary to compile, load, and use the `PuzzleMuncher` program. *Note:* if these programs depend on external library code other than the Java JDK or the `gnu.getopt` suite, the submission tarball **MUST** either include the library whole or provide easy and explicit instructions on how and where to access such libraries. This documentation **MUST** be provided in the `README.TXT` file. The designer is responsible for ensuring that all copyright and distribution conditions are adhered to.

`README.TXT` This file **MUST** describe how to compile, configure, and install the `PuzzleMuncher` program. It **MUST** also list any dependencies on additional software support libraries.

Internal documentation The handin **MUST** also include the full, compiled JavaDoc documentation for all Java source files in the submission tarball. This documentation **MUST** include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by the code. This documentation hierarchy **MUST** be included in a sub-directory named `documentation/` within the submission tarball package.

User documentation The handin submission **MUST** include complete user-level documentation for the `PuzzleMuncher` program. This documentation **MUST** include descriptions of how to use the program, any command line arguments, and the format of the input file. It must also describe the various `PUZZLES` and their respective `HEURISTICS`, supported by the code. It must describe the output format of the program, how to interpret results, and at least one example of input/output and interpretation for each `PUZZLE`. This document **MUST** be named `USERDOC.extension`, but it **MAY** be a plain text, HTML, PDF, or PostScript document (with the appropriate `extension`). It **MUST NOT** be a Microsoft Word or other nonportable format document.

Performance Documentation This handin **MUST** include a document describing the performance of the `PuzzleMuncher` system on different `PUZZLES` and with different `HEURISTICS`. It must clearly demonstrate differences in performance under different `HEURISTICS` and with different sizes of various `PUZZLES` and give some analysis as to why some `HEURISTICS` are better or worse than others. The designer **MAY** choose any tests that she or he desires to establish the performance of her/his `PuzzleMuncher` system, but **MUST** describe all the tests and why they lead to the stated conclusions about performance. This document **MUST** be named `PERFORMANCE.extension`, but it **MAY** be a plain text, HTML, PDF, or PostScript document (with the appropriate `extension`). It **MUST NOT** be a Microsoft Word or other nonportable format document.

Test cases The submission tarball **MUST** include a subdirectory named `tests/` that includes all of the input files used to demonstrate the performance of the `PuzzleMuncher` system.

CVS log file(s) For each Java source file, the submission tarball **MUST** include a corresponding `.log` file including the CVS log for that sourcecode.

At the programmer's option, this submission **MAY** also include:

BUGS.TXT This file documents any known outstanding bugs, missing features, performance problems, or failures to meet specifications of your submission. Note that the penalty for such problems will be smaller if they're fully documented here than if the instructors discover them independently.

6 Timeline

Feb 23 Project specification handed out.

Mar 1, 3:00 PM Milestone 1 due.

Mar 12, Midnight Milestone 2 due.

Mar 29, 3:00 PM Rollout.

All times given are Mountain time. Note: The period March 15–19 is Spring Break. Late days and slack days *will not* accumulate during Spring Break, nor during the following weekend (March 20 and 21). **However**, the days March 13 and 14 (Sat and Sun before Spring Break) *will* count toward late/slack days. Thus, there is significant incentive to complete Milestone 2 before Spring Break.

Appendix A: The A* Algorithm

A more complete description of the A* algorithm will be forthcoming in class and in a handout, but the fundamental pseudocode for the core of the algorithm is as follows:

```

function BestFirstSearch
open:=(s) // start node
closed:=()
while open≠()
  x:=Dequeue(open)
  if (Goal-p(x))
    return path-to-x
  else
    ChildList:=Children(x)
    foreach y ∈ ChildList
      y.parent:=x
      if (y∉open && y∉closed)
        Enqueue(open,y,f(y))
      else if (y∈open)
        if (f(y)≤f(Find(y,open)))
          update f and parent for Find(y,open)
        end
      else // y ∈ closed
        if (length(path-to-y) <
          length(path-to-Find(closed,y)))
          Dequeue(closed,Find(y,closed))
          Enqueue(open,y,f(y))
        end
      end
    end
    Enqueue(closed,x)
  end
return FAIL
end

```

Figure 2: Pseudo-code for the A* algorithm.