The goal of this project is to write programs that solve a Tile Puzzle. An example is shown in Figure 1. On the left is a starting puzzle which is a grid of tiles. Each tile contains 4 triangles that each have one of several colors. Each tile can spin around its center. In a solved puzzle, the tiles are rotated so that all adjacent triangles along the rows and columns have matching colors. On the right in Figure 1 is a solved puzzle corresponding to the starting puzzle on the left. The size of a tile puzzle can vary in either dimension.

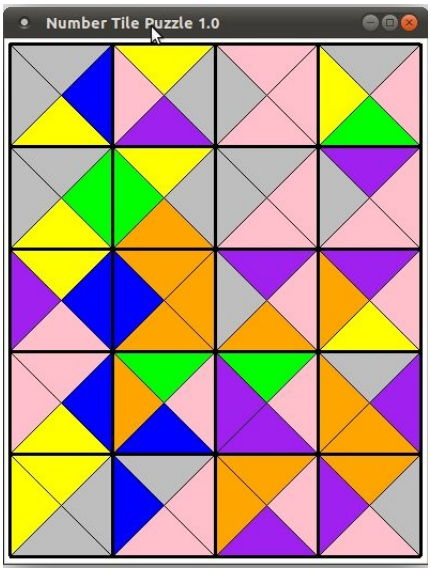
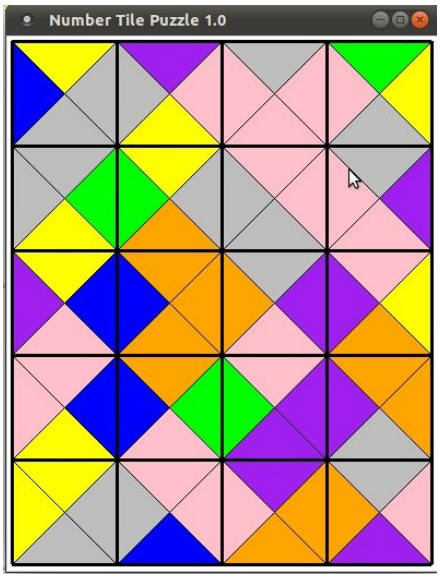


Figure1: Starting Puzzle



Solved Puzzle

Strategy: Unlike many "function only" programming tasks where a solution can be quickly envisioned and implemented, this task requires a different strategy.

1. Before writing any code, reflect on the task requirements and constraints. *Mentally* explore different approaches and algorithms, considering their potential performance and costs. The metrics of merit are **static code length**, **dynamic execution time**, and **storage requirements**. There are often trade-offs between these parameters. Sometimes *back of the envelope* calculations (e.g., how many comparisons will be performed) can help illuminate the potential of an approach.

2. Once a promising approach is at hand, an high-level language (HLL) implementation can deepen its understanding. Adding loop counters, print statements, or better still, automatic execution analysis tools (e.g., gprof) can give a more accurate estimate of performance and efficiency. HLL implementation (in C for this project) is more flexible and convenient for exploring the solution space. The final version will be completed in machine assembly where changes, big and small, are much more costly. The C version is submitted for P1-1.

3. Once a final C version is selected, it time to "be the compiler" and see how it translates to MIPS assembly. This is an opportunity to see how HLL constructs are supported on a machine platform (at the ISA level). This level requires the greatest programming effort; but it also uncovers many new opportunities to increase performance and efficiency. While assembly coding is less common in system design, it will let you see how the "rubber hits the road". The assembly version is submitted for P1-2.

P1-1: High Level Language Implementation:

In this section, the first two steps described above will be completed. It's fine to start with a simple implementation that produces an answer; this will help deepen your understanding. Then experiment with your best mentally-formed ideas for a better performing solution. Use any instrumentation techniques that help to assess how much work is being done. Each hour spent exploring here will cut many hours from the assembly level implementation exercise.

Required Files:

You will be provided with the following three files:

makefile

puzzle_backend.o

puzzle_solver.c

You can only modify puzzle_solver.c (and **rename it to P1-1.c for submission**). Inside puzzle_solver.c, you must write a complete **solver** routine such that your program can successfully complete the game.

The backend will initialize the puzzle randomly, and then call **solver**(...) with four parameters. The **solver** method has the following prototype:

void **solver**(int row_size, int column_size, int color_num, int *packed_puzzle, int *solution);

row_size: number of rows in the puzzle

column_size: number of columns in the puzzle

color_num: number of colors

packed_puzzle: an integer array of size row_size*column_size, stored in row-major lay out. Each element specifies how the corresponding cell is colored, using the following coding scheme:

North	West	South	East
Byte 3	Byte 2	Byte 1	Byte 0

Figure 2: Order in which triangle colors are packed.

solution: an integer array of size row_size*column_size. It is already allocated by the backend. Your program should return the solution in this array.The array is also stored in row-major order, where each element specifies the number of clockwise 90-degree turns you would like to apply to the corresponding cell. For example, if you set solution[0] to 1, then it means your solution spins the upper-left cell clockwise by 90 degrees.

When the **solver**(...) function returns, it returns to the backend, which will then validate whether the solution is correct or not.

Compiling/Running your code:

To compile: put all 3 files in the same directory, and simply enter 'make' as a command.

To run: (after you have compiled your code), enter ./puzzle (a) (b) (c) (d) (e)

where

(a) is the number of rows in the puzzle, for example, 4.

(b) is the number of columns in the puzzle, for example, 6.

(c) is the number of colors to be used, for example, 6.

(d) is the random seed, for example, 134. You may find it helpful to reuse a seed in order to recreate the same puzzle and debug specific scenarios. However, your program will ultimately be expected to handle any seed thrown at it. Enter **-1** to test with a different random seed each time.

(e) is the 'verbose' toggle. 1 is to print, 0 does not print.

Submission:

When have completed the assignment, submit the single file `P1-1.c` to the class website. Although it is good practice to employ a header file (e.g., `P1-1.h`) for declarations, external variables, etc., in this project please include this information at the beginning of your submitted program file. For your solution to be properly received and graded, there are a few requirements:

1. The file must be named **`P1-1.c`**. *Do not submit the executable* **`P1-1`**.
2. Your program must compile and run with the provided makefile under Linux.
3. **Your solution must be properly uploaded to the submission site before the scheduled due date, 9 pm on Wednesday, 27 February 2013.**

Grading:

Your code will be compiled and subject to 100 random runs with random puzzle sizes (row_size < 6, column_size < 8, color < 9).

Coding style (well formatted and commented):	5%
Program compiles without any errors:	5%
Program does not crash:	5%
Program terminates in less than 2 second when verbose is 'off':	10%
Program successfully solves all 100 puzzles:	75%
Penalties:	
Program fails to compile:	-100%
Program crashes:	-10% each occurrence
Program executes longer than 2 second:	-10% each occurrence
Program solves a puzzle incorrectly:	-2% each occurrence

You will never earn negative points, for example, if your program triggers -200% penalty, you will earn 0%.

Additional notes:

Watch the solution that you will return from **solver**(...), it's preallocated to row_size * column_size. Be careful not to overflow the array in **solver**(...).

Tips for success:

- Solve several puzzles yourself until you are familiar with solution strategies.
- Attempt to write out your logic for solving the puzzle as though you were instructing a friend on how to solve it.
- Translate this logic into computer code.
- Be sure to start early! This project may appear deceptively easy.

You should design, implement, and test your own code. Any submitted project containing code not fully created and debugged by the student constitutes academic misconduct.

Project Grading: The project weighting will be determined as follows:

part	*description*	*due date 9 pm*	*percent*
P1-1	Tile Puzzle Solver (C implementation)	Wed., 27 February 2013	30
P1-2	Tile Puzzle Solver (Assembly)	Wed., 13 March 2013	
	correct operation, technique & style		30
	static code size		10
	dynamic execution length		25
	operand storage requirements		5
	total		100

Project Submission Instructions:

To submit your project, you will upload the answer to each part of the assignment as a separate file. When you are ready to submit a file as your answer to a part of the assignment, use your web browser to go to the web site: http://www.ece.gatech.edu/~linda/2035/projects/index.html and click on FILE UPLOAD, provide your user id and the file to upload.

Double check that the name of the file uploaded is the one you intended to submit.

If you submit a file as the answer to part of the project and later you would like to submit an improved answer, you may submit the more recent version of the file. Only the most recent one will be graded. However, versions submitted after the due date will not be graded.

Good luck and happy coding!