This project explores the design and implementation of a constrained graph search algorithm. The program finds the lowest cost path of fixed length from a starting square (pictured in green below) to an edge square (pictured in yellow below) on a weighted map. Each square has a unique index (blue number) given as the effective linear array address of the 2-D array coordinates (`11 * row + column`), where `(row,column)` is shown in red. The gray areas are outside the map, as will be explained below.

(0,0) 0 10000	(0,1) 1 10000	(0,2) 2 10000	(0,3) 3 10000	(0,4) 4 10000	(0,5) 5 -99	(0,6) 6 10000	(0,7) 7 10000	(0,8) 8 10000	(0,9) 9 10000	(0,10) 10 10000
(1,0) 11 10000	(1,1) 12 10000	(1,2) 13 10000	(1,3) 14 10000	(1,4) 15 -49	(1,5) 16 -37	(1,6) 17 83	(1,7) 18 10000	(1,8) 19 10000	(1,9) 20 10000	(1,10) 21 10000
(2,0) 22 10000	(2,1) 23 10000	(2,2) 24 10000	(2,3) 25 -64	(2,4) 26 19	(2,5) 27 67	(2,6) 28 -82	(2,7) 29 -2	(2,8) 30 10000	(2,9) 31 10000	(2,10) 32 10000
(3,0) 33 10000	(3,1) 34 10000	(3,2) 35 -30	(3,3) 36 -37	(3,4) 37 64	(3,5) 38 -68	(3,6) 39 92	(3,7) 40 40	(3,8) 41 2	(3,9) 42 10000	(3,10) 43 10000
(4,0) 44 10000	(4,1) 45 27	(4,2) 46 -70	(4,3) 47 52	(4,4) 48 79	(4,5) 49 -77	(4,6) 50 -31	(4,7) 51 -41	(4,8) 52 -77	(4,9) 53 -81	(4,10) 54 10000
(5,0) 55 80	(5,1) 56 -73	(5,2) 57 -8	(5,3) 58 -59	(5,4) 59 53	(5,5) 60 0	(5,6) 61 -47	(5,7) 62 46	(5,8) 63 -32	(5,9) 64 98	(5,10) 65 -98
(6,0) 66 10000	(6,1) 67 13	(6,2) 68 -99	(6,3) 69 -47	(6,4) 70 -60	(6,5) 71 -14	(6,6) 72 -45	(6,7) 73 80	(6,8) 74 15	(6,9) 75 69	(6,10) 76 10000
(7,0) 77 10000	(7,1) 78 10000	(7,2) 79 -9	(7,3) 80 8	(7,4) 81 -88	(7,5) 82 69	(7,6) 83 54	(7,7) 84 12	(7,8) 85 53	(7,9) 86 10000	(7,10) 87 10000
(8,0) 88 10000	(8,1) 89 10000	(8,2) 90 10000	(8,3) 91 -21	(8,4) 92 3	(8,5) 93 -61	(8,6) 94 -43	(8,7) 95 29	(8,8) 96 10000	(8,9) 97 10000	(8,10) 98 10000
(9,0) 99 10000	(9,1) 100 10000	(9,2) 101 10000	(9,3) 102 10000	(9,4) 103 45	(9,5) 104 -34	(9,6) 105 -85	(9,7) 106 10000	(9,8) 107 10000	(9,9) 108 10000	(9,10) 109 10000
(10,0) 110 10000	(10,1) 111 10000	(10,2) 112 10000	(10,3) 113 10000	(10,4) 114 10000	(10,5) 115 59	(10,6) 116 10000	(10,7) 117 10000	(10,8) 118 10000	(10,9) 119 10000	(10,10) 120 10000

Finding the lowest cost path through a graph is a common challenge in many applications, such as navigation (finding a path from one point to one of many possible goals) and optimization of manufacturing plans (e.g., choosing optimal sequence of design/construction steps). While it can be an expensive process in general, often constraints of the graph structure simplify the search.

Pruning the search for tractability: In this project, we'll focus on exiting a diamond grid (as above, but shown below tilted 45 degrees counterclockwise). The weights between points (in black text) are randomly generated integers between -99 and 99, inclusive. Non-participating array cells (the padding) have weights set to 10,000. Note that the linearized array offset is shown in blue in each square. **The goal is to find the minimum cost path between the center**

and any exit edge where cost is the sum of all weights encountered along the path. All paths must have a length of five squares not counting the center square. (The problem is considerably more difficult if this path requirement is relaxed.)

In an example application, the squares might be geographical locations and the weights are costs for traversing elevation changes (e.g., energy required); an autonomous vehicle might need to go from the start location to any of the edge squares with the lowest total cost but only has time to take a small finite number of steps.

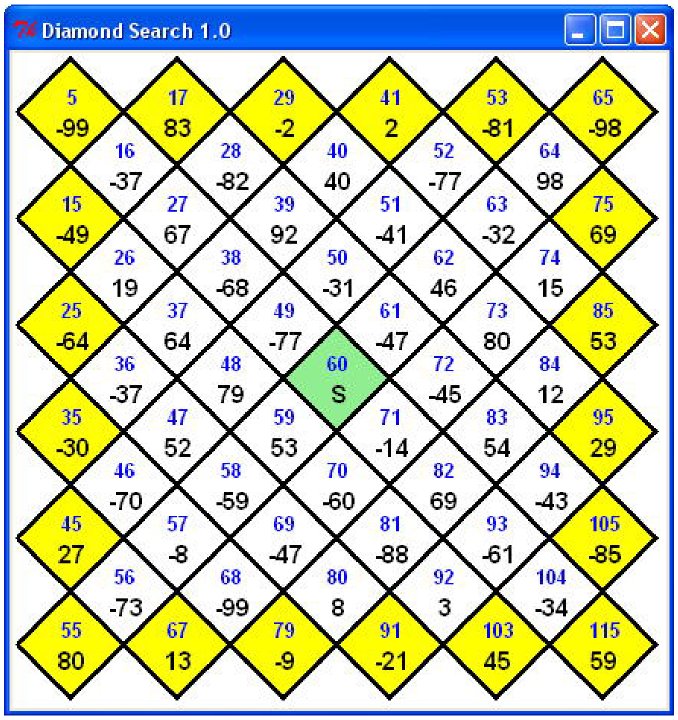

Figure 1: Diamond graph with lowest cost 5-step path of -307 (test-307.txt)

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.
2. Once a promising approach is chosen, a high-level language (HLL) implementation (e.g., in C) can deepen its understanding. The HLL implementation is more flexible and convenient for exploring the solution space and should be written before constructing the assembly version where design changes are more costly and difficult to make. For P1-1, you will write a C implementation of the program.
3. Once a working C version is created, it's 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. You will write the assembly version for P1-2.

4. *You'll hand in intermediate drafts P1-2-int.asm and the final version P1-2 at staggered due dates. All assembly files should contain a change log as described below.*

P1-1 High Level Language Implementation:

In this part, the first two steps described above are 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 ideas for a better performing solution. Each hour spent exploring here will cut many hours from the assembly level coding exercise.

You should use the simple shell C program that is provided `P1-1-shell.c` to allow you to read in a diamond graph as an array. The shell program includes a reader function `Load_Mem()` that loads the values from a text file. Rename the shell file to `P1-1.c` and modify it by adding your code.

The shell C program includes code that reads graph data in from an input file. A few sample puzzles have been provided, with the answer given in the filename (e.g., the minimum cost in "`test-307.txt`" is -307, which is the puzzle shown above). You should not rely solely on these given test files and instead create more of your own. Since manually generating input diamond graph arrays is complex, it is best to fire up Misasim, generate a diamond graph, step forward until its array is written in memory, and then dump memory to a file.

The shell C program also includes an important print statement **used for grading** (please don't change). If you would like to add more print statements as you debug your code, please wrap them in an if statement using a `DEBUG` flag – an example is given in the shell program – so that you can suppress printing them in the code you submit by setting `DEBUG` to 0. *If your submitted code prints extraneous output, it will be marked incorrect by the autograder.*

You can modify any part of this program. Just be sure that your completed assignment can read in an arbitrary diamond graph test case, compute the length of the shortest path, and correctly complete the print statement since this is how you will receive points for this part of the project.

Note: you will not be graded for your C implementation's performance (speed and storage efficiency). Only its accuracy and good programming style will be considered (e.g., organizing and commenting your code). Your MIPS implementation might not even use the same algorithm, although it's much easier for you if it does.

When you have completed the assignment, submit the single file `P1-1.c` to Canvas. You do not need to submit data files. Although it is good practice to employ a header file (e.g., `P1-1.h`) for declarations, external variables, etc., in this project you should just include this information at the beginning of your submitted program file. In order 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 worry if Canvas appends a version number.)
2. Your name and the date should be included in the header comment.
3. Your functional program should complete by printing out the minimum cost using the following print string `"The shortest path cost is [%d]\n"`.
4. Your submitted file should compile and execute on an arbitrary diamond graph (produced from Misasim). It should contain the unmodified print statement, giving the

solution. The command line parameters should not be altered from those used in the shell program.

5. Your program must compile and run with gcc under Linux. Compiler warnings will cause point deductions. If your program does not compile or enters an infinite loop, it will earn 0 correctness points.

6. Your solution must be properly uploaded to Canvas before the scheduled due date. The canvas P1-1 assignment has details on late penalties and policies.

P1-2 Assembly Level Implementation: In this part of the project, you will write the performance-focused version of the DiamondSearch program in MIPS assembly. A shell program (`P1-2-shell.asm`) is provided to get you started. Rename it to `P1-2.asm.`

Your program will call a software interrupt (**swi 514**) that will generate a random diamond graph array as a padded two dimensional array of size 11 by 11 stored in memory. Think of the diamond above as part of a larger 11 x 11 grid (like the one shown at the beginning of this document) tilted 45 degrees counterclockwise.

Your code should store the cost of the minimum path in **$4** and then call **swi 591**.

Library Routine: Here are the specifications of the three library routines you will use (accessible via the swi instruction).

SWI 514: **Create Diamond**: This routine initializes memory beginning at the specified base address with 121 entries representing the incoming weights and padding values for the 11 x 11 array. A pop up window graphically represents the map.

> INPUTS: $1 should contain the base address (memory should already be allocated).
> OUTPUTS: 121 values stored in memory. *The generated map is mutable (i.e., your program may modify it).*

The entire memory contents can be dumped to a file using the "Dump" menu button. If this is performed immediately following execution of the Create Diamond SWI, the resulting map can be read into the C version to establish the weight array using the Load_Mem function in `P1-1-shell.c`.

SWI 591: **Minimum Cost Path**: This routine allows you to specify the cost of the shortest path that your code has identified.

> INPUTS: $4 should contain an integer. This answer is used by an automatic grader to check the correctness of your code.
>
> OUTPUTS: $3 gives the correct answer. You can use this to validate your answer during testing. If you call swi 591 more than once in your code, only the first answer that you provide will be recorded.

Performance Evaluation:

In this part (P1-2), correct operation and efficient performance are both important. The assessment of your submission will include functional accuracy during 100 trials and performance and efficiency. The code size, dynamic execution length, and operand storage requirements are scored empirically, relative to a baseline solution. The baseline numbers for this project are static code size: **84** instructions, dynamic instruction length: **823** instructions (avg.), total register and memory storage required: **12** words (not including dedicated registers $0, $31, and not including the 121 words for the input weight array). *As a safety net, the dynamic instruction length metric is the maximum of the baseline metric (823 instructions) and the average dynamic instruction length of the five fastest student submissions.*

Your score will be determined through the following equation:

$$PercentCredit = 2 - \frac{Metric_{Your\,\mathrm{Pr}\,ogram}}{Metric_{Baseline\,\mathrm{Pr}\,ogram}}$$

Percent Credit is then used to scale the number of points for the corresponding points category. For example, if your program uses half as much storage as the baseline, then PercentCredit = 1.5 and the number of points for the storage category (see Project Grading table below) is 10 scaled by 1.5 = 10*1.5 = 15.

Important note: while the total score for each part can exceed 100%, especially bad performance can earn *negative credit*. The sum of the combined performance metrics scores (code size, execution length, and storage) will be capped at zero; the sum of that portion of the grade will not be less than zero points. **Finally, the performance scores will be reduced by 10% for each incorrect trial (out of 100 trials). You cannot earn performance credit if your implementation fails ten or more of the 100 trials**.

In MIPS assembly language, small changes to the implementation can have a large impact on overall execution performance. Often trade-offs arise between static code size, dynamic execution length, and operand storage requirements. Creative approaches and a thorough understanding of the algorithm and programming mechanisms will often yield impressive results. Almost all engineering problems require multidimensional evaluation of alternative design approaches. Knowledge and creativity are critical to effective problem solving.

In order for your solution to be properly received and graded, there are a few requirements.

1. Checkpoint1: A first draft version of your code should be submitted in a file named `P1-2-first-draft.asm`. This is due before the final version and will not be graded for accuracy or efficiency. It must contain a *change log*: a brief description of changes made from P1-2-shell.asm to this version of code. If this code does not incorporate substantive changes made to the shell code, points will be deducted from P1-2.

   Here are some example entries:

   ```
   # CHANGE LOG: brief description of changes made from P1-2-shell.asm
   # to this version of code.
   # Date Modification
   # 09/22 Looping through squares from start to one of the corners (example)
   # 09/23 Computing the cost of a path from start to that corner (example)
   ```

2. Checkpoint2: No additional intermediate drafts need to be submitted, but about one week before the final due date, a short graded survey will ask you to describe your progress.

3. The final version file must be named **`P1-2.asm`**. It must also contain a change log that records a brief description of changes made from the previously submitted intermediate draft to this version.

4. Your final program must produce and store the correct value in $4 and call SWI 591 to report an answer.

5. Your final program must return to the operating system via the `jr` instruction. *Programs that include infinite loops or produce simulator warnings or errors will receive zero credit*.
6. Your intermediate and final solutions must be properly uploaded to Canvas before the scheduled due dates.

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

part	*description*	*percent*
P1-1	Diamond Search (C code) correct operation, technique & style	25
P1-2	Diamond Search (MIPS assembly)	
	checkpoints, correct operation, proper commenting & style	25
	static code size	15
	dynamic execution length	25
	operand storage requirements	10
	total	*100*

All code (MIPS and C) must be documented for full credit.

Honor Policy: In all programming assignments, you should design, implement, and test your own code. Any submitted assignment containing non-shell code that is not fully created and debugged by the student constitutes academic misconduct. You should not share code, debug code, or discuss its performance with anyone. Once you begin implementing your solution, you must work alone.

Good luck and happy coding!