**The Problem**

The problem was solving an 8-puzzle tile game given two initial states and one goal state. In order to solve this problem, team members were asked to implement the A\* algorithm in C++ and run it using an unique admissible heuristic that each team member was required to develop.

The initial states that were given were the following:

Initial state 1: Initial state 2:

2 8 3 2 1 6

1 6 4 4 0 8

0 7 5 7 5 3

Goal state:

1 2 3

8 0 4

7 6 5

**The Domain**

The focus of the project was artificial intelligence and algorithm analysis. The project involved designing an admissible heuristic to use alongside the A\* algorithm in the language C++ to determine an optimal path to solve the 8-puzzle game. Project intends to evaluate the performance of each algorithm using different metrics, such as execution time, the number of nodes generated and expanded, depth of tree, effective branching factor and total path.

**Methodologies**

The program was written using a modular design approach. The code is divided into small functions, each of which handles a distinct task. Modular design was chosen because it organizes the code, making it easier to understand, maintain, and edit. This code also uses data structures such as the PuzzleNode which is a struct that contains the state (g), heuristic value (h), the total cost (f = g + h), and a pointer to the parent node. It also contains *container\_hash* which is a custom hash function that handles *std::vector<std::vector<int>>* that is used to represent the states in the *unordered\_set.* The code also takes user input to choose the initial state and heuristic function. It then outputs the solution alongside the performance metrics.

A\* Search [FINAL]

This is an informed search algorithm that is used for finding the shortest path between initial and a goal node. It initializes the open list with the start node and sets it to g and h. While the open list is full, it picks the node with lowest total cost (f') and removes it from the open list. If this node happens to be a goal node, it will then reconstruct the path and return it as a solution. It then proceeds to generate the successors of the nodes. For each of the successor, it calculates its g, h, and f cost and adds that to the closed list. This algorithm is guaranteed to find the optimal path if the heuristic used is admissible.

Diagonal conflict

This heuristic is based on identifying conflicts in tile positions along the diagonal line. It compares the position of a tile in its current state to their goal state along a diagonal line. This heuristic counts the number of tiles that are in their incorrect position and increments the counter called *conflict.*

**Source Code Implementation**

The code provided is the A\* search algorithm along with different heuristics implemented in C++ . It consists of several functions that help solve the 8-puzzle game. These include:

* *distance()*: Calculates the distance between the current state and goal state
* *diagonal \_conflict()*: Unique heuristic that calculates the number of diagonal conflicts in the current state.
* *is \_goal()*: Checks if current is the goal state.
* *generate\_successors()*: Generates the successor by shifting tiles to empty spaces
* *print\_puzzle()*: Prints the sliding puzzle
* *create\_puzzle\_node()*: Creates a new node with given state, goal state, and heuristic choice.
* *extract \_path()*: Prints the solution path
* *a\_star()*: The main function that implements the A\* search algorithm.
* *get\_nodes()*: Returns the number of nodes generated.
* *get\_nodes\_expanded()*: Returns the depth of the tree.
* *get\_b()*: Calculates the effective branching factor
* *get\_total\_path():* Calculates the total path

The code starts off with the libraries that are used:

* <iostream >: Used to output to users, and to read the user's input.
* <vector>: Used to build a dynamic array that allows the element to be accessed efficiently using indices. Building block of the puzzle tiles.
* <queue>: Used to implement the open list in the A\* search algorithm. It sorts the components based on a supplied comparator. It manages the PuzzleNodes depending on their overall cost (f = g +h).
* <unordered\_set>: Implements the closed list in the A\* search algorithm. This allows for quick insertion and lookup of visited states.
* <algorithm>: Used for the reverse function (std::reverse(path.begin(), path.end())) that is called in the extract path function. It reverses the order of the path vector so that states go from initial to goal state.
* <chrono>: Used to measure the execution time.

It then defines the PuzzleNode struct to represent the puzzle state, the cost (g), heuristic value (h), its total cost (f') and a pointer to its parent node.

The *create\_puzzle\_node* function sets node's g value to 0, h value is set to whatever heuristic the user chooses, the f' value is set to h' + 0. We then set CLOSED to the empty list.

In the *a\_star function*, the OPEN and CLOSED list is initialized. *Piority\_queue* is used to store the nodes in the OPEN list, and *unordered\_set* is used to store nodes in the CLOSED list. A while loop is opened in order to iterate the code until a goal node is found or the OPEN list is empty. If the OPEN list is not empty, it searches for the *best\_node* by using *open.top()*. While this is done, the program records the number of nodes expanded and returns it to the function *get\_nodes\_expanded*. Once the node with the lowest f' value is found, the program checks to see if *best\_node* is a goal node using the *is\_goal()* function. If it is a goal state, the node is removed from the OPEN list and is added to the CLOSED list. If not, it generates successors of the current state. The successors are generated in the *generate\_successors()* function. An if/else statement is opened to check if the successor state is in the closed list or not. If the successor is in the closed list, the algorithm will try to find the corresponding state using the *std::find\_if* function. If the state is found another function called the *propagate\_cost\_downward* function to update the cost of the nodes along the path in a depth-first manner. This will only be done if the new cost is lower than the current cost of the node. Otherwise, the successor state is unexplored. If this happens, the node will be added to the OPEN list.

The *generate\_successor* function takes a 2D vector called 'state' which holds the configuration of the states. The function returns a 3D vector containing all of the possible configurations that may be reached by shifting one tile over to an empty space. The function begins by determining the dimensions of the n (board) and finding the coordinates of i0, j0. It then initializes an empty 3D vector that holds the possible moves: up, down, left, and right. Then it iterates through each possible move calculating its coordinates. Finally, it checks to see if the new coordinates are within the board boundaries. If it is, it will create a copy of the current state and swap the values with the new coordinates. The result is then added to the successor vector.

In the main function of the program, users are asked which initial state they would like to solve. They then are asked to pick a heuristic from the following:

* Diagonal Conflict
* A\* Search Final
* Orderly

*main()* then executes the A\* algorithm, measures its performance, and outputs the results. If user does not choose a valid option, program will then run initial state 1 using the A\* Search Final algorithm.

*diagonal\_conflict()* function works by representing the number of diagonal conflicts in the current state of the board when compared to the goal state. It takes two 2D vectors as input, the state and the goal. Begins by initializing the integer *conflict\_count*. This integer will be used to count the number of conflicts found. It then determines the size of the board using the *state* vector. After that, it will iterate through all of the elements using a nested loop. For each of the elements, it will make sure the value is not 0. If this is true, it will calculate the coordinate of the value in the goal state. Then it will proceed to check to see if the element's position is in a diagonal conflict with its position in the goal state. If there is conflict, it increments the counter. After all this is done, the function will return the total number of conflicts found multiplied by 2. This is done to penalize these conflicts in the heuristic algorithm.

*distance()* function is used to calculate the distance between the initial state of the tile and the goal state. It initializes an integer called distance to 0. Gets the size of the board by using

*state.size().* Like the diagonal conflict, it uses nested loops to iterate and get the state of each tile. It then calculates the goal of position by using the formula: goal = ( value - 1) / n for the row and

goal = (value - 1) % for the column. It then computes the distance between the positions by using the formula abs(i- goal\_i) + abs(j - goal\_j), and adds this to the distance. Once all the have been computed, it returns the total distance.

*orderly()* function initializes the away variable to 0. It then defines the size of the puzzle labeled as n, and number of columns labeled as state. A 2D vector with different weight for the board. The weights are then multiplied by the distance of tiles in those positions.

**Source Code**

**#include <iostream>**

**#include <vector>**

**#include <queue>**

**#include <unordered\_set>**

**#include <algorithm>**

**#include <chrono>**

**int nodes\_generated = 0;**

**int nodes\_expanded = 0;**

**struct PuzzleNode {**

**std::vector<std::vector<int>> state;**

**int g;**

**int h;**

**int f;**

**PuzzleNode\* parent;**

**};**

**int distance(const std::vector<std::vector<int>>& state, const std::vector<std::vector<int>>& goal) {**

**int distance = 0;**

**int n = state.size();**

**for (int i = 0; i < n; ++i) {**

**for (int j = 0; j < n; ++j) {**

**int value = state[i][j];**

**if (value != 0) {**

**int goal\_i = (value - 1) / n; // row**

**int goal\_j = (value - 1) % n; // column**

**distance += abs(i - goal\_i) + abs(j - goal\_j);**

**}**

**}**

**}**

**return distance;**

**}**

**std::vector<std::vector<int>> tile\_weights = {**

**{1, 2, 3},**

**{4, 5, 6},**

**{7, 8, 9}**

**};**

**//Implement unique heuristic here**

**int diagonal\_conflict(const std::vector<std::vector<int>>& state, const std::vector<std::vector<int>>& goal) {**

**int conflict\_count = 0;**

**int n = state.size();**

**for (int i = 0; i < n; ++i){**

**for (int j = 0; j < n; ++j){**

**int value = state[i][j];**

**if (value != 0) {**

**int goal\_i = (value - 1) / n;**

**int goal\_j = (value - 1) % n;**

**if ((i - j == goal\_i - goal\_j || i + j == goal\_i + goal\_j) && (i != goal\_i || j != goal\_j)) {**

**conflict\_count++;**

**}**

**}**

**}**

**}**

**return conflict\_count \* 2;**

**}**

**int orderly(const std::vector<std::vector<int>>& state, const std::vector<std::vector<int>>& goal) {**

**int away = 0;**

**int n = state.size();**

**std::vector<std::vector<int>> weights =**

**{**

**{10, 9, 8},**

**{3, 0, 7},**

**{4, 5, 6}**

**};**

**for (int i = 0; i < n; ++i)**

**{**

**for (int j = 0; j < n; ++j)**

**{**

**int value = state[i][j];**

**int addative = weights[i][j];**

**if (value != 0)**

**{**

**int goal\_i = (value - 1) / n;**

**int goal\_j = (value - 1) % n;**

**away += addative\*(abs(i - goal\_i) + abs(j - goal\_j));**

**}**

**}**

**}**

**return away;**

**}**

**double euc\_distance(const std::vector<std::vector<int>>& state, const std::vector<std::vector<int>>& goal) {**

**double distance = 0;**

**int n = state.size();**

**for (int i = 0; i < n; ++i) {**

**for (int j = 0; j < n; ++j) {**

**int value = state[i][j];**

**if (value != 0) {**

**int goal\_i = (value - 1) / n;**

**int goal\_j = (value - 1) % n;**

**distance += sqrt(pow((i-goal\_i),2) + pow((j-goal\_j),2));**

**}**

**}**

**}**

**return distance;**

**}**

**bool is\_goal(const std::vector<std::vector<int>>& state, const std::vector<std::vector<int>>& goal) {**

**return state == goal;**

**}**

**std::vector<std::vector<std::vector<int>>> generate\_successors(const std::vector<std::vector<int>>& state) {**

**int n = state.size();**

**int i0, j0;**

**for (int i = 0; i < n; ++i) {**

**for (int j = 0; j < n; ++j) {**

**if (state[i][j] == 0) {**

**i0 = i;**

**j0 = j;**

**break;**

**}**

**}**

**}**

**std::vector<std::vector<std::vector<int>>> successors;**

**std::vector<std::pair<int, int>> directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};**

**for (const auto& direction : directions) {**

**int new\_i = i0 + direction.first;**

**int new\_j = j0 + direction.second;**

**if (new\_i >= 0 && new\_i < n && new\_j >= 0 && new\_j < n) {**

**std::vector<std::vector<int>> new\_state = state;**

**std::swap(new\_state[i0][j0], new\_state[new\_i][new\_j]);**

**successors.push\_back(new\_state);**

**}**

**}**

**return successors;**

**}**

**void print\_puzzle(const std::vector<std::vector<int>>& puzzle) {**

**for (const auto& row : puzzle) {**

**for (const auto& value : row) {**

**std::cout << value << " ";**

**}**

**std::cout << "\n";**

**}**

**}**

**PuzzleNode\* create\_puzzle\_node(const std::vector<std::vector<int>>& state, const std::vector<std::vector<int>>& goal, int heuristic\_choice) {**

**PuzzleNode\* node = new PuzzleNode();**

**node->state = state;**

**node->g = 0;**

**//node->h = distance(state, goal);**

**if (heuristic\_choice == 1) {**

**node->h = distance(state, goal);**

**} else if (heuristic\_choice == 2) {**

**node->h = diagonal\_conflict(state, goal);**

**} else if (heuristic\_choice == 3) {**

**node->h = orderly(state, goal);**

**} else if (heuristic\_choice == 4) {**

**node->h = euc\_distance(state, goal);**

**}else {**

**std::cout << "Invalid choice. Defaulting to distance." << std::endl;**

**node->h = distance(state, goal);**

**}**

**node->f = node->g + node->h;**

**node->parent = nullptr;**

**//increments node counter**

**nodes\_generated++;**

**return node;**

**}**

**std::vector<PuzzleNode\*> extract\_path(PuzzleNode\* goal\_node) {**

**std::vector<PuzzleNode\*> path;**

**PuzzleNode\* current\_node = goal\_node;**

**while (current\_node != nullptr) {**

**path.push\_back(current\_node);**

**current\_node = current\_node->parent;**

**}**

**std::reverse(path.begin(), path.end());**

**return path;**

**}**

**void print\_path(const std::vector<PuzzleNode\*>& path) {**

**for (size\_t i = 0; i < path.size(); ++i) {**

**std::cout << "Step " << i << ":\n";**

**print\_puzzle(path[i]->state);**

**std::cout << "\n";**

**}**

**}**

**struct container\_hash {**

**std::size\_t operator()(const std::vector<std::vector<int>>& c) const {**

**std::size\_t seed = 0;**

**for (const auto& row : c) {**

**for (const auto& elem : row) {**

**seed ^= std::hash<int>{}(elem) + 0x9e3779b9 + (seed << 6) + (seed >> 2);**

**}**

**}**

**return seed;**

**}**

**};**

**void propagate\_cost\_downward(PuzzleNode\* node, int g\_propagated, const std::vector<std::vector<int>>& goal, int heuristic\_choice) {**

**if (g\_propagated + node->h < node->f) {**

**node->g = g\_propagated;**

**node->f = g\_propagated + node->h;**

**for (const auto& successor\_state : generate\_successors(node->state)) {**

**PuzzleNode\* successor = create\_puzzle\_node(successor\_state, goal, heuristic\_choice);**

**if (node->parent != nullptr && node->parent->state == successor->state) {**

**continue;**

**}**

**propagate\_cost\_downward(successor, g\_propagated + 1, goal, heuristic\_choice);**

**delete successor;**

**}**

**}**

**}**

**std::vector<PuzzleNode\*> a\_star(PuzzleNode\* initial\_node, const std::vector<std::vector<int>>& goal, int heuristic\_choice) {**

**auto cmp = [](PuzzleNode\* a, PuzzleNode\* b) { return a->f > b->f; };**

**//Stores nodes in the OPEN list and is ordered by f' value**

**std::priority\_queue<PuzzleNode\*, std::vector<PuzzleNode\*>, decltype(cmp)> open(cmp);**

**//Stores nodes in the CLOSED list**

**std::unordered\_set<std::vector<std::vector<int>>, container\_hash> closed;**

**open.push(initial\_node);**

**//Iterates until a goal node is found or no solution**

**while (!open.empty()) {**

**//retrieving best\_node and removed from OPEN list**

**PuzzleNode\* best\_node = open.top();**

**nodes\_expanded++;**

**open.pop();**

**//CHECKS TO SEE IF BESTNODE IS GOAL NODE**

**//checks current state is the goal state**

**if (is\_goal(best\_node->state, goal)) {**

**return extract\_path(best\_node);**

**}**

**closed.insert(best\_node->state);**

**//generates successors of the current state**

**auto successors = generate\_successors(best\_node->state);**

**for (const auto& successor\_state : successors) {**

**if (closed.find(successor\_state) != closed.end()) {**

**continue;**

**}**

**PuzzleNode\* successor\_node = create\_puzzle\_node(successor\_state, goal, heuristic\_choice);**

**//set best\_node to successor**

**successor\_node->parent = best\_node;**

**successor\_node->g = best\_node->g + 1;**

**//compute g & f'**

**successor\_node->f = successor\_node->g + successor\_node->h;**

**if (closed.find(successor\_state) != closed.end()) {**

**auto it = std::find\_if(closed.begin(), closed.end(), [&successor\_state](const auto& old\_state) {**

**return old\_state == successor\_state;**

**});**

**if (it != closed.end()) {**

**propagate\_cost\_downward(successor\_node, best\_node->g + 1, goal, heuristic\_choice);**

**}**

**} else {**

**open.push(successor\_node);**

**}**

**}**

**}**

**return {}; // Return an empty vector if no solution is found.**

**}**

**int get\_nodes() {**

**return nodes\_generated;**

**}**

**int get\_nodes\_expanded() {**

**return nodes\_expanded;**

**}**

**int get\_tree\_depth(const std::vector<PuzzleNode\*>& path) {**

**return path.size() - 1;**

**}**

**int get\_total\_path(const std::vector<PuzzleNode\*>& path) {**

**return path.size();**

**}**

**double get\_b(const std::vector<PuzzleNode\*>& path){**

**int ng = get\_nodes();**

**int d = get\_tree\_depth(path);**

**if(d == 0){**

**return 0.0;**

**}**

**return static\_cast<double>(ng) / d;**

**}**

**int main() {**

**int heuristic\_choice;**

**int choice;**

**std::vector<std::vector<int>> initial\_state1 = {**

**{2, 8, 3},**

**{1, 6, 4},**

**{0, 7, 5}**

**};**

**std::vector<std::vector<int>> initial\_state2 = {**

**{2, 1, 6},**

**{4, 0, 8},**

**{7, 5, 3}**

**};**

**std::vector<std::vector<int>> goal\_state = {**

**{1, 2, 3},**

**{8, 0, 4},**

**{7, 6, 5}**

**};**

**std::cout << "Choose an initial state: " << std::endl;**

**std::cout << "1: " << std::endl;**

**print\_puzzle(initial\_state1);**

**std::cout << std::endl << "2: " << std::endl;**

**print\_puzzle(initial\_state2);**

**std::cout << "Enter : ";**

**std::cin >> choice;**

**std::cout << std::endl;**

**std::vector<std::vector<int>> initial\_state;**

**switch (choice) {**

**case 1:**

**initial\_state = initial\_state1;**

**break;**

**case 2:**

**initial\_state = initial\_state2;**

**break;**

**default:**

**std::cout << "Invalid choice. Using the first initial state.\n";**

**initial\_state = initial\_state1;**

**break;**

**}**

**std::cout << "Choose a heuristic: " << std::endl;**

**std::cout << "1. A\* Final" << std::endl;**

**std::cout << "2. Diagonal conflict" << std::endl;**

**std::cout << "3. Orderly" << std::endl;**

**std::cout << "4. Euclidean distance" << std::endl;**

**std::cout << "Enter: ";**

**std::cin >> heuristic\_choice;**

**std::cout << std::endl;**

**if(heuristic\_choice != 1 && heuristic\_choice != 2 && heuristic\_choice != 3 && heuristic\_choice != 4) {**

**std::cout << "Invalid choice. Using A\* Final" << std::endl;**

**heuristic\_choice = 1;**

**}**

**auto start\_time = std::chrono::high\_resolution\_clock::now();**

**PuzzleNode\* initial\_node = create\_puzzle\_node(initial\_state, goal\_state, heuristic\_choice);**

**std::vector<PuzzleNode\*> solution\_path = a\_star(initial\_node, goal\_state, heuristic\_choice);**

**auto end\_time = std::chrono::high\_resolution\_clock::now();**

**auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time).count();**

**if (!solution\_path.empty()) {**

**print\_path(solution\_path);**

**std::cout << "Solution found in " << duration << " ms:" << std::endl;**

**std::cout << "Total nodes generated: " << get\_nodes() << std::endl;**

**std::cout << "Total nodes expanded: " << get\_nodes\_expanded() << std::endl;**

**std::cout << "Depth of Tree: " << get\_tree\_depth(solution\_path) << std::endl;**

**std::cout << "Effective branching factor b\*: " << get\_b(solution\_path) << std::endl;**

**std::cout << "Total cost: " << get\_total\_path(solution\_path) << std::endl;**

**} else {**

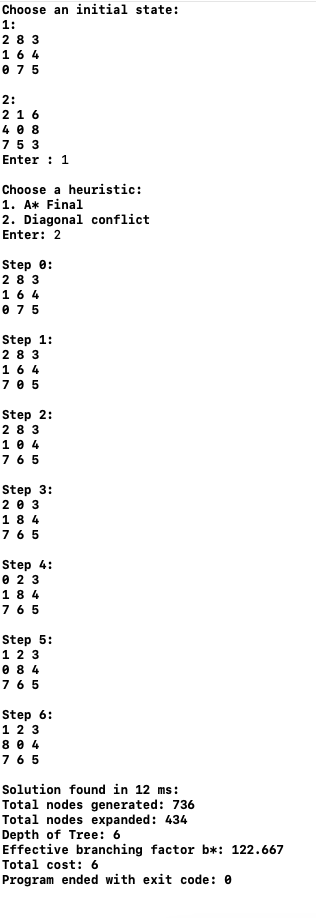
**std::cout << "No solution found." << std::endl;**

**}**

**return 0;**

**}**

**Copy of the Program Run**



**Analysis of the program**

This code is an implementation of A\* search algorithm in order to solve the 8-puzzle game. This code provides the user 2 initial states to choose from and 4 heuristics to use alongside the A\* search. It then analyzes the performance of the algorithm.

The main implementation that we did was to provide a switch, and if/else statements in order to give the user options for them to choose what initial state and what heuristic to do.

**Tabulation of Results**

Initial state 1

| Diagonal | ET | NG | NE | D | b\* | TP |
| --- | --- | --- | --- | --- | --- | --- |
|  | 15 ms | 736 | 434 | 6 | 122.667 | 7 |

| Orderly | ET | NG | NE | D | b\* | TP |
| --- | --- | --- | --- | --- | --- | --- |
|  | 7 ms | 447 | 287 | 12 | 37.25 | 13 |

| A\* Final | ET | NG | NE | D | b\* | TP |
| --- | --- | --- | --- | --- | --- | --- |
|  | 2 ms | 63 | 38 | 6 | 10.5 | 7 |

| Euc | ET | NG | NE | D | b\* | TP |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 ms | 44 | 25 | 6 | 7.33 | 7 |

Initial State 2

| Diagonal | ET | NG | NE | D | b\* | TP |
| --- | --- | --- | --- | --- | --- | --- |
|  | 674 ms | 102032 | 72219 | 18 | 5668.44 | 19 |

| Orderly | ET | NG | NE | D | b\* | TP |
| --- | --- | --- | --- | --- | --- | --- |
|  | 127 ms | 12399 | 8415 | 34 | 364.676 | 35 |

| A\* Final | ET | NG | NE | D | b\* | TP |
| --- | --- | --- | --- | --- | --- | --- |
|  | 91 ms | 9459 | 6082 | 18 | 525.5 | 19 |

| Euc | ET | NG | NE | D | b\* | TP |
| --- | --- | --- | --- | --- | --- | --- |
|  | 95 ms | 10008 | 6458 | 18 | 556 | 19 |

**Analysis of the results**

Based off these results we can conclude that the A\* search algorithm is the most efficient. While Orderly heuristic does generate more nodes, the effective branch is more effective however it is everything else is less efficient.

**Conclusion**

To conclude, I had fun working on this project. I was able to go a little out of my comfort zone with C++ and got to touch on things that I hadn't worked with before. I learned about A\* search and different heuristics that could be used.

**Team Members Contributions**

* Ana: developed diagonal conflict heuristic, and assisted with building the a\* search algorithm
* Rebekah: developed unique heuristic, and assisted with building the a\* search algorithm
* Jonathan: developed unique heuristic, and assisted with building the a\* search algorithm

**References**

<https://www.geeksforgeeks.org/unordered_set-operators-in-c-stl/?ref=gcse>

<https://www.boost.org/doc/libs/1_64_0/boost/functional/hash/hash.hpp>