Friday, 19 April 2019

Artificial Intelligence Assignment 1 Documentation

General

The **checklist** from the **instructions** given can be found at the bottom of this document.

States were represented as the Puzzle class given as Skelton code for the assignment. I implemented methods as I needed them for each algorithm

Breadth-First Search Without Visited List:

Containers:
For this algorithm I used a standard Queue (std::queue) to store partial paths at runtime. This data structure provides a first in first served approach, perfect for breadth first search as this is how the paths should be expanded.

Pseudo Code:

1. Create the first Puzzle instance with the initial and goal state and add it to the queue.
2. While the queue is not empty, get the first element from the queue, expand it and add the children Puzzle states to the front of the queue. Exit when:
    1. The goal state has been taken from the queue. Return the goal (Goal found).
    2. The Q is empty. (Solution was not found), return an empty path.
    3. Cannot allocate more memory to maintain the program. (Bad::alloc), return an empty path.

Breadth-First Search With Visited:

Containers:

Again for this algorithm I used a Queue for storing partial paths for the same reasons outline above.

For the visited list I used a vector that storing strings. The reason why I choose to store strings as apposed to puzzles in the vector as they are lighter in regards to memory. I used a vector as apposed to other structures as it has random access memory.

Pseudo Code:

1. Create the first Puzzle instance with the initial and goal state and add it to the queue.
2. While the queue is not empty, get the first element from the queue, expand it and add the children Puzzle states to the front of the queue, **if they are not in the visited list**. **Add the element to the visited list** Exit when:
    1. The goal state has been taken from the queue. Return the goal (Goal found).
    2. The Q is empty. (Solution was not found), return an empty path.
    3. Cannot allocate more memory to maintain the program. (Bad::alloc), return an empty path.

Progressive Deepening Search Without Visited List:

Containers:

For this algorithm I used a standard Stack (std::stack) to store partial paths at runtime. This structure is the opposite to a queue, stacks are a first in last searched structure. This is the best for deepening search as the algorithm requires the children of a state to be expanded before states of the same level.

Pseudo Code:

1. Create the first Puzzle instance with the initial and goal state and add it to the **stack**.
2. While the **stack** is not empty, get the first element from the **stack** that is less than the max depth, expand it and add the children Puzzle states to the **back of the** queue. Exit when:
    1. The goal state has been taken from the queue. Return the goal (Goal found).
    2. The Q is empty. (Solution was not found), **increment the maxDepth and goto set 2, if maxDepth is equal to ultimate max depth exit .**
    3. Cannot allocate more memory to maintain the program. (Bad::alloc), return an empty path.

Uniform Cost Search, A* with both Manhattan and Misplaced Tiles:

Containers:

For this algorithm and A* I used a priority queue (std::priority_queue), this structure sorts when items are added. In the puzzle class I overrode the comparison operator to allow the puzzle partial paths to be sorted. The puzzle's with the lowest F Cost are always at the front of the queue.

For the expanded list I used the same structure as I have above for the listed list. I have just implemented them differently.

FCost calculation:

Uniform = path length
A* Manhattan = path length + sumOfManhattanDistance
A* Misplaced Tiles = path length + sumOfMisplacedTiles

Pseudo Code:

1. Create the first Puzzle instance with the initial and goal state and add it to the queue.
2. While the queue is not empty, get the first element from the queue, expand it **if it is not in the expanded list** and add **each child Puzzle state to it's correct position in the priority queue based of its F cost.** Exit when:
    1. The goal state has been taken from the queue. Return the goal (Goal found).
    2. The Q is empty. (Solution was not found), return an empty path.
    3. Cannot allocate more memory to maintain the program. (Bad::alloc), return an empty path.

Instructions

	Item	your assignment details	Comments

Friday, 19 April 2019

1	Names and ID numbers of Group Members	Reece Spragg 17046764		(maximum of 3 members in a group)
2	Operating System used for testing your codes	Windows 10		Note: your codes must be tested on Windows 10. The graphics engine only works on Windows.
3	Compiler used	GCC8.2		Note: gcc 8.2 is required
4	IDE used	CLion		(e.g. SublimeText 3, ScITE)
5	Complete source codes (cpp, h files), makefile	BFS_NO_VLIST	full	Indicate '**full**', if you have completed the implementation of an algorithm, or 'partial', if you are only submitting a partial implementation.
		BFS_VLIST	full	
		PDS_NO_VLIST	full	
		UNIFORM COST_EXP_LIST	full	
		ASTAR_EXP_LIST MISPLACED TILES HEURISTIC	full	
		ASTAR_EXP_LIST MANHATTAN DISTANCE HEURISTIC	full	
6	Is your program able to run with the 2 batch files given?	Somewhat. run_one.bat works as expected. run_all.bat does run, but as state (1) for breadth first search without vested list in runs in a permanent loop. I was unable to exit this code without breaking the rules of the algorithm.		indicate '**Yes**' or '**No**' (batch files: run_all.bat, run_one.bat)
7	Experiment Results in Excel Worksheet	Yes		indicate '**Yes**' or '**No**'
8	Extra work (Bonus): Enhancements/ Optimisations included	No.		(e.g. original implementation of the Heaps data structure used to represent the priority Q in the A* algorithm)

Notes:
Although I think I understand the algorithms, I found difficulties implementing them with C++, adding to that I struggled to get the run_all.bat to complete a full run as for my algorithms could take a long time to complete.