


[CSCI 104](#)
[Staff](#)
[Schedule](#)
[Homework](#)
[Labs](#)
[Exercises](#)
[Syllabus](#)
[Sections](#)
[Wiki](#)
[Resources](#)

[CSCI 104](#)

- [Homework 3](#)
 - [Skeleton Code](#)
- [Written Portion](#)
 - [Problem 1 - Heap Tracing \(10%\)](#)
 - [Part a \(4%\)](#)
 - [Part b \(6%\)](#)
 - [Problem 2 - A* Algorithm Tracing \(10%\)](#)
- [Programming Portion](#)
 - [Problem 3 - Comparator Functor Background \(0%\)](#)
 - [Problem 4 - Linked List Recursion Coding \(15%\)](#)
 - [Part 1 - Linked List Split/Pivot](#)
 - [Part 2 - Linked List Filter](#)
 - [Testing](#)
 - [Related Videos](#)
 - [Problem 5 - Stack class \(10%\)](#)
 - [Problem 6 - m-ary Heaps \(25%\)](#)
 - [Problem 7 - Car Puzzle Game with A* AI \(30%\)](#)
 - [Overview](#)
 - [Commands and Gameplay](#)
 - [Review the A* algorithm](#)
 - [Code overview](#)
 - [Class Notes](#)
 - [Other Requirements and Tips](#)
 - [Memory Allocation and Ownership](#)
 - [Sample Executions](#)
 - [Testing](#)
 - [Related Videos](#)
- [Checkpoint](#)
- [Submission Files](#)
 - [Commit then Re-clone your Repository](#)

Homework 3

- Due: See [homework page](#)
- Directory name in your github repository for this homework (case sensitive): hw3

Skeleton Code

Some skeleton code has been provided for you in the hw3 folder and has been pushed to the Github repository [resources](#). If you already have this repository locally cloned, just perform a `git pull`. Otherwise you'll need to clone it.

Written Portion

Problem 1 - Heap Tracing (10%)

You will answer several questions about Heaps in this problem.

For this problem, you must draw this by hand or using some form of drawing app and submit the results as a PDF. Name your file `q1.pdf`.

Part a (4%)

Perform the `buildHeap` (aka `makeHeap`) algorithm on the following array to create a **min-Heap** from the arbitrary array shown below. Show the state of the array as a binary tree after each iteration (call to `heapify()`) of the algorithm. (If that does not make sense review the lecture materials to review the `buildHeap` algorithm.)

- [7, 12, 8, 10, 15, 1, 6]

Part b (6%)

Draw the tree representation of the following *binary* Min Heap in its initial configuration, and after each operation. Make sure to clearly indicate each of your final answers. This is a sequence of operations where the heap after the previous operation is what you will use to perform the next operation. To receive credit for the latter operations, your results need to be correct for the earlier operations.

- Initial Configuration: [2, 4, 6, 8, 10, 12, 14, 16]
- Insert 5
- Pop (top element)
- Pop (top element)
- Insert 5

Problem 2 - A* Algorithm Tracing (10%)



You are given the above unweighted graph, and want to find the shortest path from **node L** to **node A**, using **A* Search**. Your algorithm has the following properties:

- It uses Manhattan distance to the target node for the heuristic (the h-value) and distance travelled from the source (through the predecessor nodes for the g-value)
- If two nodes look equally good, it breaks ties by selecting the node with a smaller heuristic (or, equivalently, the node with the largest distance travelled)
- If two nodes are still tied, it break ties by choosing the node which comes first alphabetically.

Place your answer in a file `q2.txt` which lists the order that nodes are explored and discovered (start from `L`) showing the `g`, `h`, and `f` values for each node as it is discovered. Use the format shown below (where the values are fictitious and intended for demonstrating the format):

```
- explore L
  - discover H (g=1, h=9, f=10)
  - discover P (g=1, h=7, f=8)
- explore P
...
```

Programming Portion

Problem 3 - Comparator Functor Background (0%)

The following is background info that will help you understand how to do the next step.

If you saw the following:

```
int x = f();
```

You'd think `f` is a function. But with the magic of operator overloading, we can make `f` an object and `f()` a member function call to `operator()` of the instance, `f` as shown in the following code:

```
struct RandObjGen {
    int operator() { return rand(); }
};
```

```
RandObjGen f;
int r = f(); // translates to f.operator() which returns a random number by calling rand()
```

An object that overloads the operator() is called a **functor** and they are widely used in C++ STL to provide a kind of polymorphism.

We will use functors to make a sort algorithm be able to use different sorting criteria (e.g., if we are sorting strings, we could sort either lexicographically/alphabetically or by length of string). To do so, we supply a functor object that implements the different comparison approach.

```
struct AlphaStrComp {
    bool operator()(const string& lhs, const string& rhs)
    { // Uses string's built in operator<
      // to do lexicographic (alphabetical) comparison
      return lhs < rhs;
    }
};

struct LengthStrComp {
    bool operator()(const string& lhs, const string& rhs)
    { // Uses string's built in operator<
      // to do lexicographic (alphabetical) comparison
      return lhs.size() < rhs.size();
    }
};

string s1 = "Blue";
string s2 = "Red";
AlphaStrComp comp1;
LengthStrComp comp2;

cout << "Blue compared to Red using AlphaStrComp yields " << comp1(s1, s2) << endl;
// notice comp1(s1,s2) is calling comp1.operator() (s1, s2);
cout << "Blue compared to Red using LenStrComp yields " << comp2(s1, s2) << endl;
// notice comp2(s1,s2) is calling comp2.operator() (s1, s2);
```

This would yield the output

```
1 // Because "Blue" is alphabetically less than "Red"
0 // Because the length of "Blue" is 4 which is NOT less than the length of "Red (i.e. 3)
```

We can now make a templated function (not class, just a templated function) that lets the user pass in which kind of comparator object they would like:

```
template <class Comparator>
void DoStringCompare(const string& s1, const string& s2, Comparator comp)
{
    cout << comp(s1, s2) << endl; // calls comp.operator()(s1,s2);
}

string s1 = "Blue";
string s2 = "Red";
AlphaStrComp comp1;
LengthStrComp comp2;

// Uses alphabetic comparison
DoStringCompare(s1,s2,comp1);
// Use string length comparison
DoStringCompare(s1,s2,comp2);
```

In this way, you could define a new type of comparison in the future, make a functor struct for it, and pass it in to the DoStringCompare function and the DoStringCompare function never needs to change.

These comparator objects are used by the C++ STL map and set class to compare keys to ensure no duplicates are entered.

```
template < class T,                                // set::key_type/value_type
          class Compare = less<T>,                // set::key_compare/value_compare
          class Alloc = allocator<T>              // set::allocator_type
        > class set;
```

You could pass your own type of Comparator object to the class, but it defaults to C++'s standard less-than functor `less<T>` which is simply defined as:

```
template <class T>
struct less
{
    bool operator() (const T& x, const T& y) const {return x<y;}
};
```

For more reading on functors, search the web or try [this link](#)

Problem 4 - Linked List Recursion Coding (15%)

In this problem you will practice implementing recursive functions that process linked lists. Skeleton code is provided in `resources/hw3`. Copy those files to your `hw-username/hw3` folder, if you have not already. There are **two separate functions** recursive functions that we will ask you to write. They are unrelated to each other (just part a, and part b of this problem), but each of the two problems below will use the following Node definition.

```
struct Node {
    int val;
    Node *next;
};
```

You may declare and use helper functions as you deem necessary. Remember to handle the cases when an input linked list is empty. Also, most recursive solutions are *elegant*. If you find yourself writing a lot of code, you likely aren't on the right track. If you had a recursive linked list **tracing** problem in a prior homework, that might give you an idea of the *elegance* we are referring to.

Part 1 - Linked List Split/Pivot

Write a **recursive** function to split the elements of a singly-linked list into two output lists, one containing the elements less than or equal to a given number, the other containing the elements larger than the number. You must **MAINTAIN the relative ordering** of items from the original list when you split them into the two output lists. The original list should **not** be preserved. Your function must be **recursive** - you will get **NO** credit for an iterative solution. It must also run in **O(n)**, where n is the length of the input list (and can be done with only one pass/traversal through the list).

Here is the function you should implement:

```
void llpivot (Node*& head, Node*& smaller, Node*& larger, int pivot);
```

When this function terminates, the following holds:

- `smaller` is the pointer to the head of a new singly linked list containing all elements of head that were less than or equal to the pivot.
- `larger` is the pointer to the head of a new singly linked list containing all elements of head that were (strictly) larger than the pivot.
- the linked list head no longer exists (head should be set to NULL).

Note: `smaller` and `larger` may be garbage when called (i.e. you canNOT assume they are NULL upon entry). Also you should not delete or new nodes, but just change the pointers to form the two other lists.

As an example, suppose the list pointed to by `head` contained 2 4 8 3. If we used 5 as the pivot and called:

```
llpivot(head, smaller, larger, 5);
```

Then:

- `head` should be an empty list
- `smaller` should contain 2 4 3
- `larger` should contain 8

See `llrec.h` for more details and description and then place your implementation in `llrec.cpp`.

Part 2 - Linked List Filter

Write a **recursive** function to filter/remove elements of a singly-linked list that meet a specific criteria. The criteria for removal is provided by a comparison (`Comp`) functor/function object that provides an `operator()` that takes in an `int` and returns a `bool` if the node should be removed/filtered. Filtered nodes should be deallocated. Your function must be recursive - you will get **NO** credit for an iterative solution. It must also run in **$O(n)$** , where n is the length of the input list (and can be done with only one pass/traversal through the list).

```
template <typename Comp>
Node* llfilter(Node* head, Comp pred);
```

As an example, if the list pointed to by `head` contained: 3 6 4 9 and the `Comp` object's `operator()` returns true for an *ODD* integer input, then the function should return a pointer to the list containing 6 4 (since all the odd integers would have been filtered out).

Since this is a templated function (to allow for different function object types), you should put your implementation in `llrec.h`. See `llrec.h` for more details and description.

Testing

We will not provide any **formal** tests for this problem. Instead, you will be required to think through the various input cases that should be tested to ensure your code works in all cases. We have provided a skeleton file `llrec-test.cpp` in `resources/hw3` with a `main()` and some helper functions that read in values from a file to create a linked list, print a linked list, and deallocate a linked list. Complete `llrec-test.cpp` to exercise your functions and verify behavior **as you see fit**. Currently, it only reads in the contents of a file and creates the corresponding linked list. You can then call your functions on that list, print results, etc. to verify the correctness of your implementation.

You must update your Makefile with a target `llrec-test` that will compile the necessary code in the various source files including `llrec-test.cpp` into an executable named `llrec-test`. Once you have compiled your test program, you can run it and provide an input file. See an example below:

```
./llrec-test llrec-test1.in
```

We have provided one input test file, `llrec-test1.in` that you can use. Feel free to create other input files and use that as input. (It would be appropriate to add/commit/push those files if you create them).

Note: We will not grade your `llrec-test.cpp` or any input files you create. They are **SOLELY** for your own benefit to test your code. **After submission**, we will test your code with our own full test suite and assign points based on those tests. But you will not have access to these tests, so you need to test your own code thoroughly.

We ask that you **NOT SHARE** your test program or input files with other students. We want everyone to go through the exercise of considering what cases to test and then implementing those tests.

Related Videos

A video overview of how these functions may work and be organized [is available here](#).

Problem 5 - Stack class (10%)

Implement a templated Stack class, `Stack<T>`.

It must:

- inherit from `std::vector<T>` and you need to choose whether public or private inheritance is the best choice. Though composition would generally be preferable since a Stack is not truly a vector, we want you to practice with templates and inheritance.
- Support the following operations with the given signatures (see header file)

```
Stack();
size_t size() const;
bool empty() const;
void push(const T& item);
void pop();
T const & top() const;
```

- If `top()` or `pop()` is called when the stack is empty, you must throw `std::underflow_error` defined in `<stdexcept>`.
- All operations **must run in O(1) time**. Failure to meet this requirement will result in **MAJORITY** of credit being deducted for this problem.
- **Important Note:** To call a *base* class function that has the same name you cannot use `this->member_func()` since both classes have that function and it will default to the derived version and lead to an infinite recursion. Instead, scope the call (e.g. `LList<T>::size()`).
- It would probably be a good idea to write a very simple test program (e.g. `stack_test.cpp`) just to ensure your code can pass some basic tests. We will not grade or require separate stack tests. You will use your stack in a later programming problem which should help you test it, but it is always good to test one component at a time.

Problem 6 - m-ary Heaps (25%)

Now that you have learned the basics of heaps, build your own m-ary Heap class whose public definition and skeleton code are provided in the `heap.h` skeleton file. Rather than specifying a specific type of heap (Min- or Max-Heap) we will pass in a Comparator object so that if the comparator object functor implements a **less-than** check of two items in the heap then we will have a min-heap. If the Comparator object functor implements a **greater-than** check of two items in the heap, we will have a max-heap.

```
template <typename T, typename Comparator = std::less<T> >
class Heap
{
public:
    /// Constructs an m-ary heap for any m >= 2
    Heap(int m = 2, Comparator c = Comparator() );

    // other stuff
```

```
};
```

You may use the STL `vector<T>` container if you wish. Of course, you **MUST NOT** use the STL `priority_queue` class or `make_heap`, `push_heap`, etc. algorithms.

Notice we can provide a default template type for the Comparator so that the client doesn't have to specify it if they are happy with the `std::less` (i.e. which assumes a built-in `operator<` is available for type `T` and calls it).

Notice the constructor also provides *default* value for `m` and the Comparator which is a default-constructed Comparator. Default parameter values will be used if the client does not supply them. Also if a default parameter is used then all parameters after it must also have default values (i.e. default parameters must be at the end of the declaration). The last thing to note is you only supply the default value in the declaration of the function. **When you define the function (i.e. write the implementation below) you would not specify the default value again but just leave the argument types/names.** For example, when implementing the function you'd just define:

```
template <typename T, typename Comparator>
Heap<T,Comparator>::Heap(int m, Comparator c /* Don't specify the default value here, only above */ )
// Your code here
```

So with these defaults in place, the client that wants a **binary min-heap** with a type that has a **less-than operator** need only write:

```
Heap<string> h1; // calls the default constructor with default args:
// m=2 and std::less<string> is used as the default template type for Comparator
// and std::less<string>() (i.e. the default constructor) will be the
// comparator object created by the constructor
```

If the client wants some custom method of comparison so that they can implement a max-heap or some other alternative using a custom comparator can write:

```
ObjAComparator c1( /* some constructor arguments as desired */ );
Heap<ObjA, ObjAComparator> h1(2, c1);
```

Before using our tests, we recommend you write a simple test program of your own to check some simple cases that you come up with. You could push a few integers in random order and then pop them all out and ensure they are in order. You may want to try heaps of different types like `int` and/or `string`. Also, be sure it works as a min-heap and as a max-heap using different comparators. For reference if your type `T` has a `<` operator, then C++ defines a `less` functor which will compare the type `T` items using the `operator<()`. Similar there is a `greater` functor already defined by C++ that will compare using the `operator>()`. They are defined in the `functional` header (`#include <functional>`) and you can look up their documentation online for further information. This is meant to save you time writing functors for types that can easily be compared using built in comparison operators.

Once you feel like your heap seems to be on the right track, you can use a subset of basic tests that we have provided (`hw3_heap_checker` folder in resources) to check your heap. They are not necessarily exhaustive and we may run more complete tests for grading but this should help you. Just copy them to your `hw3` folder, `cd hw3_heap_checker`, and run the normal `cmake ., make` and run the tests.

Note: You may use this Heap object in future homeworks, so test it well now!

Problem 7 - Car Puzzle Game with A* AI (30%)

Overview

In this application you will use the A* algorithm to “efficiently” solve a puzzle game where **vehicles** in a square 2D grid (aka the **board**) must be **moved** (by sliding) to allow a certain vehicle (vehicle a in our program) to **escape** off the right side of the board by moving right along its given row. To start, a **Board** (i.e. the 2D grid) that shows the initial layout of vehicles (each identified by a character a, b, c and so on...) will be provided and read in by the program. We guarantee vehicle a is always facing horizontally and is the vehicle we want to help **escape** by moving off the right side of the board. Vehicle a may start in any row. Other vehicles must be slid horizontally and vertically in the board to allow vehicle a to escape. Most of the examples we give will be a 6x6 board to match the classic game, but your program should work for LARGER board sizes.

If you are unfamiliar with the game, an online version is available to try (with ads...sorry) [at this site](#).

We’ve provided a majority of the code to allow the user to play a text-based version of this game on their own. Your task is to add the ability for the user to:

- Ask for a cheat where your program will use A* to find a shortest sequence of moves that will solve the puzzle and display those moves to the user for their consideration. The user can then key these in or go their own route, asking again later for another cheat sequence based on their updated board state.
- Undo the moves already entered if the user realizes those moves were not useful and have the state of the board be restored to its previous value(s) before the move(s) were made.

Commands and Gameplay

A sample starting board might look be the following:

```
...b..
...b..
aa.b.d
..cccd
.....
.....
```

Here we see the **escape** vehicle a positioned horizontally in row 2. Where a is located in the escape row (i.e. in which columns it is located) may vary. For consistency, we will number the rows and columns from 0, starting at the upper left corner.

Each vehicle will be of length 2 or more and be positioned HORIZONTALLY or VERTICALLY. The user can then choose the vehicle to slide by entering its letter **identifier** as well as a **postive or negative** amount of positions to move. We will use the covention that **positive** amounts mean **right** or **down** for horizontal and vertical vehicles, respectively, while **negative** amounts mean **left** or **up** for horizontal and vertical vehicles, respectively.

For the example above, the game can be won by moving c by an amount -2 (2 to the left), b by an amount 3 (3 down), and then d by an amount 1 (1 down) as shown below. Note: there are other potential combinations to win, but the optimal solution can be done in 3 moves.

```
...b..
...b..
aa.b.d
..cccd
.....
.....
```

Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : c -2

```
...b..
...b..
aa.b.d
ccc..d
.....
```

```

.....
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : b 3

.....
.....
aa...d
cccb.d
...b..
...b..
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : d 1
You win!

.....
.....
aa....
cccb.d
...b.d
...b..
You win!

```

Note the game ends as soon as a has an open path to its right to escape.

Review the A* algorithm

Recall that A* chooses the move with smallest f-value to explore next. Note: $f = g + h$ where g = distance (number of moves made) from the start state while h is a score produced by a heuristic evaluation of the move. Please take some time to review the algorithm presented in class (slides, notes, etc.). We will use the following heuristics:

- **Brute-force:** Returns $h=0$...this causes A* to degenerate to a Breadth-first search
- **Direct blocking vehicles:** Counts the number of vehicles to the right of the escape vehicle in its row (i.e. number of vehicles directly blocking its escape).

Ex.	Ex.
ee.c..	.ddee.
...c..	.ffc..
-> aabc..	..bc..
..b...	-> aabc.g
..b...	..b..g
.ddd..
h = 2	h = 3

Note: the -> is not printed or part of the input but just shown on this writeup to highlight the escape row.

In the first example (above and to the left), $h=2$ because b and c are in a's row and blocking it from escape. In the next example, b, c, and g are in a's row and blocking it.

- **Indirect blocking vehicles:** (Read this carefully a few times) Counts the **direct** blocking vehicles (defined just above) **AND** what we define as **indirect** blocking vehicles. At a high level, an **indirect blocking vehicle** is one that blocks a **direct blocking vehicle** from making a **necessary** move to clear the escape row. To compute **indirect blocking** vehicles, you will need to iterate through each **direct** blocking vehicle and check the following:

For a vehicle to qualify as an indirect blocking vehicle it must satisfy this criterion:

- Be above or below a *direct blocking vehicle* that has **ONLY 1** viable path to clear the escape row and is being blocked by this vehicle (i.e. the *direct blocking* vehicle's length precludes it from clearing the escape

row in one direction and the *indirect blocking* vehicle is blocking its ability to clear the escape row in the other direction).

This implies that if a *direct blocking vehicle* can move up or down to clear the escape row, then by definition there cannot be an *indirect blocking vehicle* associated with that *direct blocking vehicle*. Furthermore if the *direct* blocked vehicle has the option to move BOTH up OR down to clear the escape path but is blocked by vehicles in both directions, then we do **NOT** consider those as *indirect* blocking vehicles since it does not have **ONLY 1** viable path to clear the escape row. This somewhat counterintuitive choice to **not** count those vehicles above and below the *direct* blocked vehicle is explained later in more detail and pertains to the requirements of the heuristics we use in order for A* to produce an **optimal** solution. Let's look at some examples to illustrate what vehicles should be counted in this heuristic and which should not.

In the following examples, the direct blocking vehicles can move out of the way without requiring another vehicle to move.

Ex. 1	Ex. 2
...ee.	..d...
.....	..d...
-> aabc..	..b...
..bc..	-> aabc..
..d...	...c..
..d...	...ee.
h = 2	h = 2

In example 1, b can move up while c can move down to clear the escape row. Thus, **neither d nor e** are *indirect* blocking vehicles.

Ex. 3	Ex. 4
ee.c..	.ddee.
...c..	.ffc..
-> aabc..	..bc..
..b...	-> aabc..
..b...	..b...
.ddd..gg
h = 3	h = 5

In example 3, b nor c can move up to clear the escape row because of their length. Their only viable option to clear the escape row is to move down, but d is blocking those moves. Thus, we add d to our heuristic score to obtain **h=3** (notice we should only count d once since a single move may be sufficient to clear the path for b and c).

In example 4, neither b nor c can move down to clear the escape row due to their length, thus, they must move up, but d and f **MUST** move to allow b to clear the escape route. Thus, we count them in our heuristic. e is blocking c and so we count it as well to obtain a heuristic score of **h=5**.

The tougher case for our heuristic to count correctly arises when a *direct blocking vehicle* is blocked in both the up and down directions by another vehicle. We will **NOT** define one or the other as an *indirect* blocking vehicle **BECAUSE it is not immediately obvious which one should be moved**. For reasons you will learn if/when you take CS 360 (AI), heuristics must **underestimate** the number of moves to a solution if A* is to *guarantee* an optimal solution. When only a *single* vehicle above or below (but not both) is blocking a direct blocking vehicle, it is clear the vehicle **MUST** be moved and would not be an overestimate of the number of moves required to solve the puzzle. However, when a **direct** blocking vehicle has a vehicle above AND below that precludes it from clearing the escape row, which one to move may not be obvious and, if we choose wrong, can lead to an overestimate, thus nullifying the guarantee of an optimal solution. So, in these scenarios, we will not count **EITHER** of the vehicles above or below in our heuristic. (If one of those vehicles is the only vehicle blocking

some **other direct** blocking vehicle from clearing the escape row, we may count it when we process that **other direct** blocking vehicle).

Ex. 5	Ex. 6	Ex. 7
.ee...	.ddee.	.ee.gg
...c..	...c..	..bc.d
-> aabc..	-> aabc..	-> aabc.d
..bc..	..b...	...c.h
..b...	..ff..	..fffh
.dd...gg
h = 3	h = 2	h = 4

In example 5, we notice b is blocked in the up direction by e and in the down direction by d. However, d still meets the criteria of an **indirect** blocking vehicle because b's length would never allow it to clear the escape row by moving up in the first place. Thus, e should not influence b from identifying d as the **ONLY** blocking vehicle. Thus, the heuristic score is 3 (d and the **two direct blocking vehicles**).

In example 6, b's length does NOT preclude it from moving up or down to clear the escape row, but it is blocked in the up and down direction (by d and f) so we count neither d nor f. Similarly, c could potentially clear the escape row in either direction, but is also blocked in both directions by e and f. So we do not count either of those. Thus, the final heuristic score is 2 (the count of the **direct** blocking vehicles, b and c).

In example 7, b's length does NOT preclude it from moving up or down to clear the escape row, but it is blocked in the up and down direction (by e and f) so we count neither e nor f. When c is examined, we see that it cannot move up because of its length and f is blocking it in the downward direction, so f does meet the criteria for an **indirect blocking** vehicle and is thus counted (even though it wasn't counted when b is processed). When d is processed, we find that it is not too long to move up or down to clear the escape row, but it is also blocked in both the up and down direction (by g and h) and thus we do count neither g nor h. This leads to the heuristic value of 4 (f and the **three direct blocking** vehicles).

One last note: We could devise a way to count which vehicle to choose to move when a direct vehicles is block above and below, but heuristics should be **fast** to compute and solving which vehicle to choose to move is akin to solving the whole problem. Thus, we use our simplified approach.

Code overview

We have placed skeleton code in the resources/hw3 folder.

1. Board class - Implements the basic functionality of the game and stores the state of vehicles and grid. It provides the basic operations to move a vehicle, print the board, check if the game is solved, etc. Essentially, this class has all the basic functionality for writing a program to play the game manually (without any cheat/AI) and will provide some abilities for the cheat/AI.
2. Move class - Used as the primary data object that the A* search algorithm uses in its open- and closed-lists. Represents a possible move storing a board (with its vehicle configuration), the g and h scores of the move, a pointer to its parent move in the search tree, and the vehicle and move amount pair that led from the parent move to this move.
3. Heap<T> class - Your heap class from the earlier part of the HW. It will be used for the open list of your A* search algorithm. **In this application you should choose m=2. You must use the heap you wrote in the previous problem.**
4. Solver class - A class to implement the A* search algorithm and store the solution and statistics that the client can then retrieve after running the A* algorithm.

5. **Heur** (Heuristic Base & Derived classes - A polymorphic interface to compute the heuristic score for a Board as well as the three derived implementations (BFS, Direct, and Indirect)).
6. **Stack<T>** class - You will need to use your Stack class from the prior problem to implement some feature of this gameplay. Look for where it would be most appropriately used.
7. **Main Application** (rh.cpp) - Implements the basic gameplay loop and processes commands entered by the user.

Class Notes

You should not change the interface of any member functions unless otherwise specified. You may add other helper member functions as you deem useful.

1. **Board class:** Please refer to the comments and documentation in `board.h` which serves as requirements along with this document.

- You may not alter the public interface of the Board class but may add data members and private helpers. Some public member functions need to be completed.
- The `isLegalMove()`, `move()`, and `solved()` functions are “complete” and working (though you may update, as necessary).
- You will need to implement the `potentialMoves()`. A move is a pair of the vehicle identifier and the amount to move that vehicle (positive or negative). This function should determine all the vehicles that can move and all the different amounts they can move (positive or negative) with one potential move per combination of vehicle and movement amount. For the board below the potential moves would be:

(a,1) (c,-1) (c,-2) (d,-1) (d,-2) (d,1) (d,2)

```
...b..
...b..
aa.b.d
..cccd
.....
.....
```

- You will need to implement an `operator<()` to compare this board and another. This will allow you to use Boards as keys in a set or map to determine uniqueness. How you decide to compare boards is your choice though you can consider converting the boards to some kind of string and utilize its comparison operators.
- We have provided `operator<<` (ostream operator) to output the board to any ostream in a 2D text format.
- You will need to complete an `undoLastMove()` function to restore the previous state of the Board. Follow the documentation in the `board.h` header file.
- Other functions which you can see in the provided code.

1. Move class

- Each state in the A* search requires a Board configuration but also some additional metadata. This struct specifies this metadata. Since a struct is public by default, you can just access the data members from other code entities. But it would likely help to make some constructors and possibly a destructor. Feel free to add/change any constructors or other member functions you deem necessary.
- We have also setup two functors at the bottom of this file which you need to complete. One will be used by the open-list (heap) to compare two `PuzzleMove`'s and determine which has the smallest f-score. Important: To ensure we get the same answers we need to break ties in a consistent manner.

You must follow these rules:

- If `move1` has a smaller f-score than `move2`, we consider `move1 < move2` to be true

- If move1 and move2 have equal f-scores but move1 has a smaller h-score, then we consider move1 < move2 to be true
- If move1 and move2 have equal f-scores and equal h-scores and move1's *Board* is less-than move2's *Board*, we consider move1 < move2 to be true

The other functor is used to determine the uniqueness of two PuzzleMove's based on their boards. This will be used for the closed set so that we don't enter a duplicate move. This functor should simply compare the the first Move's board to the second's by calling the Board class' operator<() and return the result.

2. Heuristic derived classes

- So that the A* algorithm can easily use different heuristics we setup a polymorphic base class Heuristic with a pure virtual function

```
size_t compute(const Board& b)
```

This function should compute and return a heuristic score for the given board. But how it computes this will be up to the derived class. We have defined 3 heuristics described above. Implement each derived class and its compute() function according to the descriptions above. It will likely be **much easier** to define some helper functions to do subtasks as you compute the heuristic score (like finding a vehicle up or down, etc.).

3. Solver class

- This class just implements the A* search algorithm in the run(). It can be initialized with a given heuristic to be used. It also maintains a copy of the starting board (the current configuration from which the user asked for the cheat and from which we start our A* search), an appropriate data structure (your choice) to store the solution (i.e. the sequence of moves that are needed to solve the game), and the number of expansions the search algorithm took. An expansion is simply a PuzzleMove that is entered into the open-list. It roughly estimates the amount of work the algorithm performs.

At a high level your A* search algorithm will take the initial board provided, make a PuzzleMove out of it and enter it into the open list and the closed list (so we never duplicate it again). It will then start the process of dequeuing items from the open-list (i.e. a heap) one at a time, possibly generating new moves (expansions) and, if they aren't already in the closed-list, entering them into the open-list and closed-list (so we never duplicate them again). To get these potential new moves, you will take the board from the move you just dequeued from the top of the open-list (heap) and call potentialMoves(). This will return new "successor" boards around which you can construct new Moves, scoring them, setting up its previous/parent pointer, etc. and then possibly adding them to the open-list.

Think carefully about when you are ready to delete a Move. Once the solution board has been found, you need to walk the move sequence back up to the initial board, collecting which vehicles (and amounts) were moved at each step. Don't delete Moves too early. Finally, when it is time to delete moves, think carefully where they might exist (are they all in a single data structure, spread over multiple data structures, etc.). Wherever they are make sure you delete them all to avoid memory leaks.

You should be able to just implement the member functions specified in the header file. Note: we have typedef'd something we call a MoveSet. Use this type to create your closed-list (i.e. your closed-list will be a C++ set of PuzzleMove's but using the Board comparator functor to determine uniqueness). If you haven't seen a typedef before it is just an alias for a type (do some research on your own to see more examples). We usually use them to shorten long typenamees to something more readable for the user. So in this case MoveSet is simply replaced with std::set<Move*, MoveBoardComp>. Also note that when you instantiate a PuzzleMoveSet you'll need to pass in an instance of your PuzzleMoveBoardComp object that the set will use as the comparator.

If no solution exists, `solution()` should return an empty list

To debug your `run()` function, it may help to add `cout` commands to print a note when you dequeue a `Move` from the PQ (printing out its relevant info) and then add `couts` to print each successor `Move` that you generate and their relevant info. Then when you run a program you can see what your code is doing and compare it to what you know it should do (tracking it on paper, etc.).

4. Main Application

- The main application is written in `rh.cpp` and is **COMPLETE**. We accept two command line arguments:
 - `input-board-file` the text file containing the 2D board (we have provided several example boards in the `boards/` subfolder that you can use as input)
 - `heur` selection value (0=BFS, 1=Direct, 2=Indirect)
- The menu is printed on each turn. The input options are:
 - A move given by the vehicle ID and an amount (negative amounts mean up/left while positive amounts mean down/right). An example is `c -1`.
 - `?` to run the A* algorithm to find an optimal solution and display it to the user.
 - `Q` to quit the game
 - `Z` to undo the previous move (and pressing `Z` again will undo the move before that, if it exists)

Other Requirements and Tips

1. You should have no memory leaks or invalid reads/writes as checked for in Valgrind.
2. Your A* search must be implemented correctly and return a shortest solution sequence. There **MAY BE MANY POTENTIAL SHORTEST SOLUTIONS**. Any that you return is fine. Our tests will simply ensure the solution ties for the shortest and that the solution moves lead to a solved board.
3. You shall ensure your open-list (heap) compares Moves based on the algorithm for breaking ties listed above **and uses m=2**.
4. While for this particular problem it is technically fine to stop when you see the solution board/move being *added* to the openList, let us follow the general A* algorithm and stop only when we *remove/dequeue* it from the openList.

Memory Allocation and Ownership

A big learning goal of this project is to exercise your skill at managing memory appropriately. An important concept is memory ownership. It is fine for several “client” objects to each have a pointer to a dynamically allocated object (effectively sharing it). But one client object must be the **owner** of that dynamic memory and thus be responsible for deallocating it when no longer needed (i.e. deleting it in its destructor). Other client objects may have a pointer but when they are deallocated, they do NOT in turn deallocate what the pointer is pointing to. In that case, we’d say the object does NOT *own* the object being pointed to.

Take care that you only have one “owner” so that you don’t try to deallocate the same memory twice which will lead to a segfault and/or other memory errors. In addition, take care that an owning object does not mix pointers to dynamically-allocated objects and objects that live on the stack. Since your code is unable to distinguish pointers to heap-based objects vs. stack-based objects, you generally must ensure that you only have pointers to dynamically allocated objects if you are going to delete them.

Also try to use good encapsulation where the owning object will delete the memory it owns in its destructor and not make some external code do it.

Sample Executions

You may use the sample executions below to try out your program.

Sample 1 - inboard0.in (Indirect heuristic)

If we ran this command line:

```
./rh boards/inboard0.in 2
```

And then we entered ?, followed by z, followed by Q, we would see this:

```
...b..
...bee
aa.b.d
.....d
...c..
...c..
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : ?
No Solution exists!

...b..
...bee
aa.b.d
.....d
...c..
...c..
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : Z
No move to undo

...b..
...bee
aa.b.d
.....d
...c..
...c..
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : Q
Don't give up that easily!
```

Sample 2 - inboard1.in (Indirect heuristic)

Running on this input:

```
./rh boards/inboard1.in 2
```

And then entering ? should yield the solution shown, which we can then enter as moves to solve the game.

```
...b..
...b..
aa.b.d
..cccd
.....
.....
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : ?
Solution found with expansions: 19
d 2
c -2
b 3

...b..
...b..
aa.b.d
..cccd
.....
.....
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : d 2
```



```

...b..
...b..
aa.b..
..ccc.
.....d
.....d
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : c -2

```

```

...b..
...b..
aa.b..
ccc...
.....d
.....d
Enter a vehicle and move amount (+ = dn/rt, - = up/lt), 'Q', '?', or 'Z' : b 3

```

```

.....
.....
aa....
cccb..
...b.d
...b.d
You win!

```

Note: For boards/inboard1.in a **BFS** heuristic (0) yields **72** expansions and a **direct** heuristic (1) yields **30**. The **indirect** heuristic (2) yields **19** expansions. Again, there may be some variation on this number based on how you implemented your Board operator< (i.e. how ties are broken).

Testing

You can crowd source the output of your A* solutions on various boards and how many expansions were required. But for the basic boards it should be easy enough to verify the solution.

Related Videos

A video overview of some of the relevant object/class relationships and how the A* algorithm works [is available here](#).

Checkpoint

For checkpoint credit, submit your working code for the linked list recursion problem. Ensure you add/commit/push your hw-username repo with a hw3 subfolder that contains:

- Your Makefile and **all necessary source code files** so that running `make 11rec-test` will compile and create a working executable: `11rec-test` that we can test. Failure to compile will result in 0 credit for your checkpoint. There should also be no memory/Valgrind errors of any kind when we run your test on any valid input file. It is fine to push input test files if you like, though we will not grade them.
- **THEN** you must submit your SHA on our Submit page linked from the [Homework Page](#).

Submission Files

Ensure you add/commit/push all your source code files, Makefile, and written problem files. Do **NOT** commit/push any test suite folder/files that we provide from any folder other than the resources/hw3 repo.

WAIT You aren't done yet. Complete the last section below to ensure you've committed all your code.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your hw3 directory to your hw-username repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. In your terminal, cd to the folder that has your resources and hw-username
2. Create a verify-hw3 directory: `$ mkdir verify-hw3`
3. Go into that directory: `$ cd verify-hw3`
4. Clone your hw_username repo: `$ git clone git@github.com:usc-csci104-spring2022/hw-username.git`
5. Go into your hw3 folder `$ cd hw-username/hw3`
6. Switch over to a docker shell, navigate to the same verify-hw3/hw-username/hw3 folder.
7. Recompile and rerun your programs and tests to ensure that what you submitted works. You may need to copy over a test-suite folder from the resources repo, if one was provided.