# Objective

Pathfinding is a pervasive problem in computer science. What algorithm should we use to find the shortest path between two points? Is there a "best" solution to this problem? The problem can easily be reduced to a graph problem whereby an algorithm is given a starting node and end node. The graph is then traversed to find the shortest simple path between the two nodes, if such a path exists. In this project, we will explore two important subtasks used for efficient path finding: *graph traversal* and *heuristic estimation*.

# Getting Started

Before we begin, we will discuss the skeleton code provided to help you build and evaluate a complete pathfinder.

**NOTE:** A program called `maze_gen` is included with the assignment template so that you can generate your own test data.

As a warmup exercise, we need to implement a couple of support functions. The supporting utilities we need for our pathfinding algorithms are:

1. Implement a priority queue.
2. Implement three distance metrics amenable to a two-dimensional grid.

A *priority queue* is an abstract data type for an ordered set which supports the following operations:

1. **FIND** an element with the highest priority;
2. **DEQUE** an element with the highest priority;
3. **ENQUEUE** an element with an assigned priority.

The priority queue that must be implemented will be an extension of the doubly linked list in Assignment 1. The function prototypes you must implement are:

```
typedef struct pqueue_t {
    dlist_t* dl;
} pqueue_t;

pqueue_t* pqueue_new ();
int pqueue_isempty (pqueue_t *pq);
void pqueue_enqueue (pqueue_t *pq, void *item, int size);
void pqueue_enqueue_with_priority (pqueue_t *pq, void *item, int size,
                                   int (*cmpfptr)(const void*,const void*));
void pqueue_free(pqueue_t *pq, void (*delfptr)(void*));
void* pqueue_dequeue(pqueue_t *pq);
void* pqueue_find(pqueue_t *pq, void *item,
                  int (*cmpfptr)(const void*,const void*));
void* pqueue_remove(pqueue_t *pq,void *item,
                    int (*cmpfptr)(const void*,const void*));
```

Perhaps the only unclear function is `pqueue_enqueue_with_priority()`. This function is conceptually equivalent to `insert_in_order()`, but the ordering is imposed by the priority of the element being inserted.

Next, you should implement three distance metrics in `libpath{h|c}`. The three metrics are:

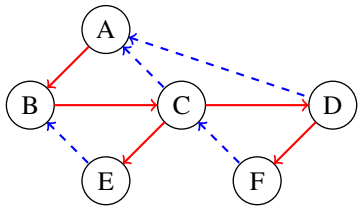
1. Euclidean Distance - Given a set of points using Cartesian coordinates $(x, y)$, the distance between two points $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is the Euclidean distance $d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.
2. Manhattan Distance - Given a set of points using Cartesian coordinates $(x, y)$, the distance between two points $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is the Manhattan distance is $d = |x_i - x_j| + |y_i - y_j|$.
3. Chebyshev (Chessboard) Distance - Given two directional vectors, the Chessboard distance is the greatest difference along all coordinate distances. Given $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$, the chessboard distance is $d = \max(|x_j - x_i|, |y_j - y_i|)$.

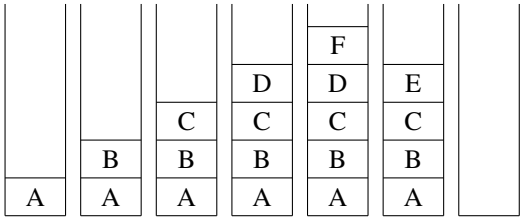
For each of these tasks, you should create a simple unit test. These will compile into the binaries `test_pqueue` and `test_dmetric`. There are no hard requirements on what your unit test should do, but they should test the functionality just as the simple unit tests provided in Assignment 1 did. Writing simple unit tests for every piece of new functionality is considered good programming practice. If you wish to practice *test driven development*, then you should write the unit test with the expected outcome **before** implementing the functions.

## TBasic Graph Traversal

Many graph algorithms require a method to process *all vertices* in a *systematic* fashion. Similar to systematic processing of trees — preorder, inorder and postorder — there are two principal algorithms for traversing a graph: **depth-first search** (DFS) and **breadth-first search** (BFS).

**Depth First Search** (DFS)**:** Depth-first search starts visiting vertices of a graph at an *arbitrary* vertex by marking it as visited. Once a vertex has been marked, the algorithm proceeds to an **unvisited** adjacent vertex if any exist. When no more unvisited adjacent vertices can be found, the algorithm backtracks along the traversal path and processes any remaining vertices marked as **unvisited** along the way. The algorithm *halts* after reaching the starting vertex. Consider the following example:



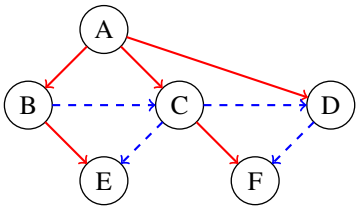




A vertex is **pushed** on the stack when the vertex is visited the first time and **popped** of the stack when it becomes a dead end.

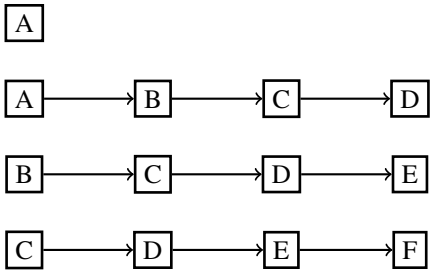
DFS EXAMPLE

The algorithm starts at an arbitrary vertex $A$, marks it as **visited**, and proceeds to the first unvisited vertex $B$. The algorithm continues to walk through the graph until it reaches vertex $F$. After processing $F$ the algorithm finds no adjacent vertex that is marked as unvisited. The algorithm therefore starts to **backtrack** along the original path until an unvisited adjacent vertex is found. vertex $E$ is visited after the algorithm backtracks to vertex $C$. The algorithm **terminates** after backtracking to the starting vertex $A$, where **all** adjacent vertices are marked as **visited**.
A **stack** can be used to keep track of the path through the traversal graph.

**Breadth-first search** (BFS)**:** Breadth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as visited. Once a vertex has been marked, the algorithm proceeds to **all** unvisited adjacent vertices of the current vertex. When no unvisited adjacent vertices exist, it moves along to process **all** adjacent vertices of the vertices marked as visited in the previous step. The algorithm halts after all vertices in a **connected component** have been visited. Consider the following example:




Queue:




**BFS EXAMPLE**

The algorithm starts at an arbitrary vertex $A$, marks it as **visited** and proceeds to **all** unvisited adjacent vertices $B$, $C$ and $D$. After processing $B$, $C$ and $D$ the algorithm starts to process **all** adjacent unvisited vertices of $B$, $C$ and $D$. The algorithm stops once all vertices are marked as visited.
A **queue** can be used to keep track of the path through the graph. The queue is initialized with the starting vertex $A$. At each iteration the algorithm identifies **all unvisited** vertices of the vertex at the front of the queue, marks them as visited, and adds them to the queue. The front item in the queue is then removed and the next front element is processed.

Our next task is to implement simple BFS and DFS graph traversal with one special condition. Traversal should always begin with a starting node provided, and terminate when the destination node is discovered. Nodes will be defined using an $xy$ coordinate system. The $xy$ coordinates can easily be derived from the graph node identifier using the length and width of the two dimensional maze. More precisely, `x = id % graph->width` and `y = id / graph->width`. For all of our implementations, we will use a `bitvector` to manage the `visited` array.

The maze and corresponding graph can be generated using `gen_maze`. The `gen_maze` program can be used as follows:

```
$ ./gen_maze
USAGE: ./gen_maze [options]
    -w maze width
    -h maze height
    -n number of routes to generate
    -d draw generation of maze <width/height == screen size>
    -e graph edge output
    -r graph route output

EXAMPLE: ./gen_maze -w 640 -h 480 -n 20 -e test.graph -r test.routes
```

The order, grid length, and grid width are written to the `test.graph` file, followed by the graph edges. The `test.routes` file contains $n$ start - destination node pairings. If you would like to find out more on how the maze generation works, see `http://en.wikipedia.org/wiki/Maze_generation_algorithm#Randomized_Prim.27s_algorithm`

There are two additional main wrappers to use in your assignment: `pathtime` and `pathview`. The `pathview` program provides a graphical interface using curses so that you can see how the algorithms work. The `pathtime` program is a simple command line wrapper to empirically evaluate the algorithms.

The usage for each is:

```
$ ./pathtime
USAGE: ./pathtime [options]
    -e graph input file
    -r routes input file
    -a algorithm (dfs/bfs/bestfirst/astar)
    -d distance metric (euclidian/manhattan/chessboard)
    -h display usage information

EXAMPLE: ./pathtime -e test.graph -r test.routes

$ ./pathview
USAGE: ./pathview [options]
    -e graph input file
    -r routes input file
    -a algorithm (dfs/bfs/bestfirst/astar)
    -d distance metric (euclidian/manhattan/chessboard)
    -h display usage information

EXAMPLE: ./pathview -e test.graph -r test.routes
```

**NOTE :** You should call the function `graph_visit()` everytime you visit a node in all of the traversal algorithms so that the visual representation works properly. If you do not call `graph_visit()`, `pathview` will not work properly.

## Task 3 - Implementing Best-First Search (3/15 marks)

For our first two search algorithms, **BFS** and **DFS** the distance metric should have no affect on the path traversal. However, finding the target node blindly is not necessarily the most effective approach. So, we would like to use a heuristic to ensure that the next node that is selected moves us closer to the final target. This is know as **best-first search**. We will implement our best-first search as a modification to **BFS**. The general idea is to allow our distance metric to guide the next choice. The pseudocode for best-first search is:

ALGORITHM **Greedy_Best_First** $(G, i, j, \delta)$
// Find the shortest path in a graph $G$ from node $i$ to node $j$.
// INPUT : A starting vertex $i$, a target vertex $j$, and a distance metric $\delta$.
// OUTPUT : The shortest simple path for $i$ to $j$.

```
1:  PQueue ← ENQUEUE(i, δ(i, j))
2:  Visited[i] ← true
3:  while PQueue ≠ ∅ do
4:      v ← DEQUEUE(PQueue)
5:      Visited[v] ← true
6:      if v = j then
7:          return
8:      end if
9:      for v' ∈ V adjacent to v do
10:         if not Visited[v'] then
11:             PQueue ← ENQUEUE(v', δ(v', j))
12:         end if
13:     end for
14: end while
```

## Implementing A* Search

The A* algorithm is a combination of Dijkstra's algorithm and a best first search. The idea is simple. Calculate the distance from a starting node $i$ ($g(n)$) and calculate the distance from the target node $j$ ($h(n)$). The total cost is $f(n) = g(n) + h(n)$. Node ordering selection will be based on a priority queue ordered by total cost $f(n)$. In addition, we need to maintain two sets `open` and `closed / visited`. For our purposes `open` is a bitvector and a priority queue, and `visited` is a bitvector and a linked list. The auxiliary bitvector is useful since **MEMBER**, **INSERT**, and **DELETE** can be performed in constant time. The `open` set contains candidate nodes to be examined. The `visited` set maintains previously examined nodes. Initially, `open` will contain only the starting node. We now pull the first item off of the priority queue. If this node is the goal, we terminate. Otherwise, we move the node from `open` to `visited`. Now, we must examine each neighbor of the node. If the neighbor is in `visited`, we check to see if the current weight can be lowered. If so, move it to `open`. If the neighbor is already in `open`, we don't need to do anything now. Otherwise, we add the neighbor to `open`. When all of the neighbors have been examined, we pull the next node from the priority queue and try to find the next best move. The pseudocode for our A* algorithm is:

ALGORITHM **A*** $(G, i, j, \delta)$
// Find the shortest path in a graph $G$ from node $i$ to node $j$.
// INPUT : A starting vertex $i$, a target vertex $j$, and a distance metric $\delta$.
// OUTPUT : The shortest simple path for $i$ to $j$.
// Note: $g(v)$ refers to the actual path length from i to $v$.

```
1:  Visited ← ∅
2:  PQueue ← ENQUEUE(i, d)
3:  while PQueue ≠ ∅ do
4:      v ← DEQUEUE(PQueue)
5:      Visited[v] ← true
6:      if v = j then
7:          return
8:      end if
9:      for v' ∈ V adjacent to v do
10:         if not Visited[v'] then
11:             d = g(v) + δ(v, v') + δ(v', j)
12:             if ⟨v', d'⟩ ∈ PQueue and d < d' then
13:                 remove PQueue(v')
14:             end if
15:             PQueue ← ENQUEUE(v', d')
16:         end if
17:     end for
18: end while
```

There are many tradeoffs associated with the data structures used to support the key operations in A*. Everyone should read the excellent discussion on A* and pathfinding at `http://theory.stanford.edu/~amitp/GameProgramming/`. This is arguably the most valuable resource you'll find on pathfinding and the algorithms you will be implementing for this assignment. **DO NOT** blindly reproduce the examples and source code found on this site (or some other, we know how to use Google pretty well too). Rather, use it to guide your intuition on how to implement the algorithm for yourself.

## Comparing the results

We now have four different pathfinding algorithms implemented. For a bonus of up to 3 additional marks, empirically compare the algorithms. You should modify the basic timing in `pathtime` to report an average time, standard deviation, and median. Support several runs to guarantee the accuracy of your measurements. Measure time and the number of nodes each algorithms traverses. Write a one page document in ASCII text called `report.txt` and compare and contrast the different algorithms based on the metrics above (or any other good ways you discover to compare the algorithms).
**NOTE 1 :** All submissions should be in ANSI C, and use no functions outside of the C standard library and maths library.

**NOTE 2 :** All files submitted in ASCII format (source code, makefiles, shell scripts) must have a maximum line width of 79 characters, so that they can be printed on A4 paper without line wrapping. You are required to submit a `Makefile` that works on a Unix machine. You code should compile using `gcc -W -Wall -ansi` and have **no**

warnings. Compiler warnings are a good indicator that something was done improperly in the implementation. A short summary of good programming practices includes:

- Check return values for `malloc, calloc, realloc` and all file I/O operations. If the error is recoverable, continue. Otherwise, `assert` and / or `exit`.
- Your code should always do sanity checking on user provided input. If your program expects an English text file, you should try to ensure this is the case.
- Use consistent indentation.
- Limit the length of each line in the program to a maximum of 79 characters.
- Use intelligent code comments.
- Avoid magic numbers.

**NOTE 3 :** In order to implement `pqueue_dequeue`, you will need some sort of "remove element from end" functionality. This can either be done in `liblist` or in `libpqueue`.