COMPUTER ENGINEERING DEPARTMENT

CMPE304 GRAPH THEORY

A* Pathfinding Algorithm

Submitted by

Samed Tok

Instructor

Aslı Tuncer

Faculty of Engineering and Natural Sciences
Kadir Has University
Spring 2025

TABLE OF CONTENTS

ABSTRACT

A* (A-star) is a widely used graph traversal and pathfinding algorithm known for its completeness, optimality, and optimal efficiency . This report presents a detailed examination of the A* algorithm, including its mathematical formulation and use of heuristic functions. We discuss conditions for heuristic *admissibility* (never overestimating the true cost) and how these conditions guarantee that A* finds an optimal path. A full complexity analysis is provided, covering best-case, average-case, and worst-case time complexity in Big-O notation, as well as memory requirements. We include a proof sketch illustrating why an admissible heuristic ensures A*'s optimality. A* is also compared with other pathfinding algorithms such as Dijkstra's algorithm and Breadth-First Search (BFS) in terms of functionality, performance, and theoretical efficiency. Empirical results are provided to illustrate these comparisons – for example, on a real road network scenario, A* expands significantly fewer nodes than Dijkstra for the same target, due to the guidance of its heuristic. Diagrams of node exploration and comparison tables are included to visualize A*'s advantages. The report concludes with an appendix containing Python code listings for a sample A* and Dijkstra implementation, and a references section with scholarly citations.

1 INTRODUCTION

Pathfinding is a fundamental problem in computer science and AI, with applications ranging from robotics and route planning to video games and network routing. The objective is typically to find the shortest or lowest-cost path between two points (nodes) in a graph. Classical graph search algorithms like **Breadth-First Search (BFS)** and **Dijkstra's algorithm** can solve shortest path problems but may become inefficient on large or complex graphs. **A*** (pronounced "A-star") is an *informed* search algorithm that improves efficiency by using a heuristic function to guide the search towards the goal . First published in 1968 by Hart, Nilsson, and Raphael , A* was developed as an extension of Dijkstra's algorithm with the addition of heuristics to prioritize promising paths. A* quickly became popular due to its ability to find optimal paths while often exploring far fewer nodes than uninformed searches.

This technical report provides a comprehensive overview of the A* pathfinding algorithm. We begin by explaining the A* algorithm's mechanics and mathematical formulation (the use of a cost function $f(n) = g(n) + h(n)$). We then discuss the role of the heuristic function $h(n)$, defining **admissible heuristics** and conditions under which A* guarantees an optimal solution. A complexity analysis is presented to characterize the algorithm's performance in best, average, and worst cases. We also sketch a proof of optimality for A* with an admissible heuristic. Next, we compare A* with two other well-known pathfinding approaches – BFS and Dijkstra – highlighting differences in functionality and efficiency. An experimental case study on a real road network (Küçükçekmece, Istanbul) is included to empirically demonstrate how A* outperforms Dijkstra in practice by exploring fewer nodes to reach the same destination. Finally, we conclude with observations on A*'s applicability and include an appendix with the Python implementation used for the simulation.

2 METHODOLOGY

2.1 Algorithm Description and Formulation

The A* algorithm searches for the shortest path from a given start node to a goal node in a weighted graph. It maintains a set of *open* nodes (frontier) to explore, prioritized by an estimate of the total path cost through each node. Formally, for each node *n*, we define:

- **g(n)**: the cost of the path from the start node to *n* (often called the "distance so far").
- **h(n)**: the heuristic estimate of the cost from *n* to the goal (an informed guess).
- **f(n)**: the estimated total cost of a path going through *n* to the goal, defined as **f(n) = g(n) + h(n)** .

At each step, A* picks the node in the open set with the lowest *f(n)* value to explore next. This node (call it *current*) is removed from the open set and added to the *closed* set (the set of already evaluated nodes). Then, all neighbors of *current* are examined. For each neighbor *m*, a tentative cost *gtentative = g(current) + cost(current, m)* is computed, where *cost(current, m)* is the weight/distance of the edge from *current* to *m*. If this tentative cost is lower than the best known *g(m)*, we update *m*'s records: set *g(m) = gtentative*, update *f(m) = g(m) + h(m)*, and set *m*'s predecessor to *current*. If *m* was not already in the open set, it is added. This process continues until the goal node is chosen for expansion (meaning the shortest path to the goal has been found), or until the open set is empty (meaning no path exists).

Pseudocode for A* (high-level):

```
function A*(start, goal):
   open_set = { start }              // nodes to explore
   closed_set = ∅                    // nodes already explored
   g[start] = 0
   h[start] = heuristic_estimate(start, goal)
   f[start] = g[start] + h[start]

   while open_set is not empty:
      current = node in open_set with lowest f(current)
      if current == goal:
```

```
        return reconstruct_path(current)   // goal reached

    open_set.remove(current)
    closed_set.add(current)

    for each neighbor m of current:
        if m in closed_set:
            continue  // already evaluated

        tentative_g = g[current] + cost(current, m)
        if m not in open_set:
            open_set.add(m)
        else if tentative_g >= g[m]:
            continue  // not a better path

        predecessor[m] = current
        g[m] = tentative_g
        h[m] = heuristic_estimate(m, goal)
        f[m] = g[m] + h[m]

    return failure  // no path found
```

In this algorithm, the heuristic_estimate(n, goal) function computes the heuristic *h(n)*, and reconstruct_path(current) backtracks via the stored predecessors to build the final path. The efficiency of A* comes from always expanding the seemingly most promising node next (lowest *f*), which directs the search toward the goal.

Admissible Heuristic and Conditions for Optimality

The choice of heuristic *h(n)* is crucial. A* is *complete* (it will find a solution if one exists, assuming finite graph or non-negative edge costs) and *optimal* (it finds the shortest path) **if the heuristic is admissible**, meaning *h(n) never overestimates the true cost to reach the goal*. In other words, for all nodes *n*, **h(n) ≤ h**(n)**,** where *h**(n)* is the actual shortest path cost from *n* to the goal . An admissible heuristic is by definition a lower bound on the remaining cost. Common examples of admissible heuristics include straight-line distance in a road network

(which never overestimates the true road distance), or Manhattan distance in a grid (for grid movements restricted to horizontal/vertical steps).

A stricter condition is **consistency (or monotonicity)**. A heuristic is *consistent* if for every node *n* and every neighbor *m* of *n*, the estimated cost obeys the triangle inequality: **h(n) ≤ cost(n, m) + h(m)** for all edges (n, m). A consistent heuristic is always admissible . Consistency guarantees that *f(n)* values along any path are non-decreasing, which in turn ensures that once a node is expanded (closed), it has found the optimal path to that node . In practical terms, consistency prevents the need to revisit nodes: A* will not need to re-open a closed node if the heuristic is consistent. Most admissible heuristics used in pathfinding (like distances in metric spaces) are also consistent. If a heuristic is admissible but *not* consistent, A* can still find an optimal solution, but it may end up expanding some nodes multiple times (increasing the run time) .

Optimality: With an admissible heuristic, A* is guaranteed to find a shortest-path solution, i.e., it is an *admissible search algorithm*. The intuition is that A* will not prematurely overlook the optimal path. If there were a shorter path to the goal than the one A* finds, A*'s heuristic (being admissible) would have caused the algorithm to explore that shorter path first. More concretely, when the goal node is finally chosen for expansion from the open set, its *f(goal)* equals *g(goal)* (since h(goal)=0) and this *g(goal)* is the shortest path distance. We can sketch a proof by contradiction: assume A* finds a suboptimal goal path with cost **C** while the true shortest path cost is **C*** < C. At the moment when the goal was selected from the open set, there must have been some node *n* on the optimal path still in the open set (because the optimal path had not been fully explored yet). For that node *n* on the optimal path, consider its *f(n) = g(n) + h(n)*. Along the optimal path, *g(n)* is the cost from start to *n*, and since the heuristic is admissible, *h(n)* ≤ remaining cost from *n* to goal. Therefore, *f(n)* ≤ cost(start→*n*) + cost(*n*→goal) = C*. Thus *f(n)* ≤ *C***. Meanwhile the suboptimal goal had *f(goal) = g(goal) = C*. Because C* < C, we have *f(n) < C*. This implies there was at least one node in the open set (such as *n*) with *f* value less than *f(goal)* at that time. A* would have chosen that node *n* for expansion before the goal, contradicting the assumption that the goal was expanded while a cheaper path was still available. *Therefore, A cannot pick a suboptimal goal when using an admissible heuristic; it will always find the optimal path** . (For a formal proof, see Hart et al. 1968 or standard AI textbooks.)

In summary, an admissible heuristic ensures A*'s optimality because it never leads the algorithm to underestimate the remaining cost. If the heuristic is also consistent, it further guarantees that each state is expanded at most once, making the search more efficient.

2.2 Complexity Analysis of A*

We analyze the time and space complexity of A* in terms of V (number of vertices/nodes) and E (number of edges) in the graph, as well as in terms of typical search tree parameters (branching factor b and solution depth d).

- **Worst-Case Time Complexity:** In the worst case, A* may have to expand a large portion of the search space. Without a good heuristic (for example, if $h(n) = 0$ for all nodes, which reduces A* to Dijkstra's algorithm), the algorithm effectively performs an exhaustive search similar to breadth-first or uniform-cost search. The worst-case time complexity is often given as **O(b^d)** (exponential in the depth d of the optimal solution) . This corresponds to the scenario where the heuristic provides no guidance and the algorithm potentially explores all nodes up to the goal depth. However, when analyzed in terms of graph size V and E, A*'s worst-case running time with an efficient priority queue (for the open set) is **O((V + E) log V)**, which is the complexity of Dijkstra's algorithm using a binary heap . In a graph, |E| is often on the order of |V| b (branching factor times number of vertices), so O((V+E) log V) can be roughly O(E log V). For example, in a dense graph this is ~O(V^2), and in a sparse graph it's closer to O(V log V). Thus, one can say *A's worst-case time complexity is on the order of Dijkstra's**. In fact, if the heuristic is admissible, A* **never expands more nodes than Dijkstra would** on the same problem . (Dijkstra is essentially A* with h=0, so any *positive* heuristic guidance cannot cause A* to do more work than Dijkstra; it will instead prune some searches and *at worst* perform the same work as Dijkstra.)
- **Best-Case Time Complexity:** In the best case, the heuristic is extremely accurate (ideally, $h(n)$ equals the true remaining cost $h^*(n)$ for all nodes). If the heuristic is *perfect*, A* will essentially march straight from the start to the goal, expanding only the nodes along the optimal path. In this ideal scenario, the time complexity is **O(d)**, where d is the number of steps in the shortest path (since each step along the path is expanded once). Even with a slightly less-than-perfect heuristic, as long as the heuristic is **very close** to the true cost, A* will expand considerably fewer nodes than in the worst case. It will largely focus on the vicinity of the optimal path. For instance, if no unnecessary

branches are explored, complexity approaches linear in the path length. In Big-O notation, one could say the best-case complexity is **O(d)** (or O(V) in the trivial case that the path goes through most of the nodes, but then that is also the minimum needed to verify the path). In practice, best-case occurs when the goal is immediately recognized or when the search space is trivial (e.g., start = goal, which is O(1)). Generally, A*'s performance is highly sensitive to the accuracy of *h(n)*.

- **Average-Case Time Complexity:** Average-case analysis of A* is complex because it depends on the heuristic's quality and the graph's structure. In typical applications with a well-designed heuristic, A* runs much faster than an uninformed search by significantly reducing the number of explored nodes. We can say that *on average, A runs in polynomial time for many practical problems**, since the heuristic effectively guides the search (for example, in a 2D grid or roadmap with a Euclidean heuristic, A* will mostly expand nodes in the direction of the goal). However, in the theoretical worst scenario (or with a poorly informative heuristic), the search may still approach the exponential upper bound. Thus, average-case performance lies between the two extremes: better than uninformed searches, but not as fast as an oracle. It is common to describe A* as *optimally efficient* over all *admissible* algorithms for a given heuristic: no other algorithm using the same heuristic can expand fewer nodes on *average* . This optimal efficiency means that A* makes the best use of the heuristic's information on typical problem instances.
- **Space Complexity:** One major drawback of A* is its memory usage. The algorithm stores all generated nodes in memory (in the open or closed sets), which in the worst case can also be exponential in the search depth. The space complexity is often given as **O(b^d)** , same as the time complexity in the worst-case, because A* could potentially store a fringe of the size of the search tree frontier. In graph terms, the space complexity is O(V) in the worst case (it might have to keep all vertices in memory if the search covers the whole graph) . This is significantly higher memory usage than, say, depth-first search (which only needs to store a path stack). For large graphs or state spaces, A* can run out of memory before running out of time. There are *memory-bounded* variants of A* (such as IDA*, SMA*) that trade off some optimality or completeness to cap memory usage, but standard A* is memory-intensive.

In summary, **A***'s complexity is **exponential in the worst case** (no good heuristic guidance), but with a decent heuristic it performs much closer to linear or polynomial time in practice. Its

time cost will never exceed that of an uninformed Dijkstra search on the same graph , and typically it will be significantly less. The trade-off is that A* requires enough memory to hold all explored nodes, which can be a limiting factor for very large problems.

2.3 Proof Sketch: Admissible Heuristic ⇒ Optimality of A*

(This section provides a brief proof outline for why an admissible heuristic guarantees that A finds an optimal path. Readers already familiar with this concept may skip to the next section.)*

As stated earlier, a heuristic is **admissible** if it never overestimates the remaining cost to the goal: ∀n, h(n) ≤ true_cost(n, goal). We want to show that if A* uses an admissible h, then when A* terminates and returns a path, that path is the optimal (shortest-cost) path to the goal.

Consider the moment when A* terminates and has chosen the goal node *G* for expansion (thus identifying the solution path). Let *G*'s cost found by A* be f(G) = g(G) + h(G). Since *G* is the goal, h(G) = 0, so f(G) = g(G), which is the cost of the path A* found. Now suppose, for the sake of contradiction, that this cost is not optimal. Let *C* = g(G) be the cost of the solution A* found, and assume there exists another route from the start to the goal with lower cost *C*** (the optimal path cost). At the time A* picked *G* from the open set, all nodes with f lower than f(G) should have been expanded already (A* always expands the lowest f). If a better path exists, there must be some node *N* on that optimal path that had not yet been expanded (otherwise the whole optimal path would be known). Consider the *first* such node along the optimal path that was still in the open set. The start node has g(start)=0 and is on the optimal path, so such an *N* exists (at least the start itself initially). Because *N* is on the optimal path, the true cost from start to *N* is g(N) (which A* will have computed by the time it inserted *N* into open set), and the true cost from *N* to goal is some t. Because the heuristic is admissible, h(N) ≤ t (it may underestimate, but not overestimate). Therefore f(N) = g(N) + h(N) ≤ g(N) + t = true_cost(start→N→goal). But true_cost(start→N→goal) ≤ C* (since it's part of the optimal path to goal). Hence f(N) ≤ C*. Now since we assumed C* < C, we get f(N) < C. This means *N* had f value less than f(G) (which was C). Thus when *G* was chosen for expansion, *N* should still have been in the open set with a smaller f, and A* would have selected *N* before *G*. This contradiction shows that our assumption (that a better path existed than A* found) is false. Therefore, A*'s solution cost C

must equal the optimal cost C*. In more formal terms: *if h is admissible, A is guaranteed to return a least-cost path from start to goal** .

This argument ensures that **A*** is an *optimal* algorithm (in terms of finding shortest paths) under the admissibility condition. Note that if h were not admissible (i.e., it overestimates somewhere), A* could fail to explore an actually optimal path because it might deem it too expensive (due to an inflated h value) and favor a suboptimal route. That is why using non-admissible heuristics can cause A* to find non-optimal solutions .

Additionally, if the heuristic is **consistent**, one can strengthen the proof by showing that whenever A* expands a node, the cost g(n) is the lowest possible and will not be improved later, so each node is expanded at most once and in increasing order of true cost. This is a more stringent proof found in many AI textbooks. However, the admissible heuristic alone is sufficient for optimality (at the possible cost of re-expanding some nodes if the heuristic is inconsistent) .

In conclusion, the admissibility of the heuristic is the key condition that makes A* both complete and optimal: it ensures A* never “misses” the optimal solution. This property is what distinguishes A* from greedy algorithms that use heuristics but can make suboptimal choices.

3 A* vs OTHER PATHFINFING ALGORITHMS

In this section, we compare A* with two fundamental pathfinding algorithms: **Breadth-First Search (BFS)** and **Dijkstra's algorithm**. We examine their differences in terms of functionality (when they are applicable, what they compute), performance characteristics, and theoretical efficiency.

3.1 A* vs. Breadth-First Search (BFS)

Breadth-First Search is an uninformed graph search that expands nodes in waves outward from the start node, exploring all nodes at distance 1, then distance 2, and so on. BFS is **guaranteed to find the shortest path in terms of number of edges** (hops) in an *unweighted* graph (or any graph where all edges have equal weight). It is also complete (it will find a goal if one exists, given finite graph). However, BFS has no heuristic guidance – it blindly explores in all directions. This means that if the state space is large, BFS can generate an enormous number of nodes before reaching the goal. Its time complexity in a graph is O(V + E) (linear in the size of the graph), which is optimal for an uninformed search, but in terms of branching factor and depth it is O(b^d) (exponential in depth) in the worst case (because it essentially explores all nodes up to depth d). The **space complexity** of BFS is also O(b^d) in the worst case, as it stores the frontier of the current wave of nodes. In practice, BFS is *feasible* only for relatively small or shallow graphs due to its memory consumption.

When comparing BFS to A*:

- **Functionality:** BFS can only be directly applied to find shortest paths on unweighted graphs (or where each step has equal cost). If edges have weights (different costs), BFS is not guaranteed to find the true shortest-distance path – it finds the path with fewest edges, which is not the same if some edges are "longer" than others. A* handles weighted graphs naturally by accounting for actual path costs *g(n)*. In fact, A* with heuristic h=0 reduces to *Uniform-Cost Search*, which is equivalent to Dijkstra's algorithm, and in an unweighted scenario that is effectively BFS. So BFS can be seen as a special case of A* (or Dijkstra) on unweighted graphs.
- **Performance:** Without a heuristic, BFS (and Dijkstra) potentially explore a lot of irrelevant nodes. Imagine a start and goal that are far apart – BFS will exhaustively

traverse everything in a radius around the start until it finally reaches the goal distance. A* would instead, with a good heuristic, "point" itself in the right direction and largely ignore nodes that lie far away from the direct route. Thus, *A is usually much faster than BFS** on problems where a guiding heuristic is available. BFS might expand an exponentially growing fringe of nodes, whereas A* might cut through the search space. It's worth noting that BFS is actually optimal in terms of the number of nodes expanded among *uninformed* methods – no blind search can guarantee to expand fewer nodes than BFS in the worst case . But as soon as we have extra knowledge (a heuristic), A* can decisively outperform BFS by prunning large portions of the search space.

- **Optimality:** BFS will find an optimal path (minimum hops) in an unweighted graph. A* will find an optimal path (minimum total weight/distance) in a weighted graph, given admissible h. In scenarios where BFS is applicable (unweighted shortest path), one can use A* with h=0 to get the same result – though that is just BFS in disguise. With a non-zero heuristic, if the heuristic is admissible, A* will still find the optimal path but more efficiently. If the heuristic were inadmissible, A* could find a suboptimal path; BFS has no heuristic, so it cannot mislead itself – it will systematically find the true shortest by breadth expansion (albeit slowly).
- **Memory:** Both BFS and A* have high memory usage (they both keep a frontier that in worst case can be exponential in size). DFS (Depth-First Search) uses less memory but cannot guarantee shortest paths (and can even get stuck in loops without extra checks). A* shares BFS's memory drawback; as F. Z. Zulfiqar notes, "the true disadvantage of [BFS] (and by extension, of A*) is that you need to keep track of all open nodes, consuming a lot of memory if the target is far away" . This is an important consideration: on very large problems, memory often runs out before time with BFS/A*.

In summary, BFS is only practical for simpler scenarios (small graphs or unit costs). A* extends the capability to handle weighted graphs and uses heuristics to dramatically improve performance. Whenever a reasonable heuristic is available, A* will find the solution significantly faster than BFS by avoiding brute-force expansion of the entire search sphere around the start.

3.2 A* vs. Dijkstra's Algorithm

Dijkstra's algorithm is the classic solution for single-source shortest paths in weighted graphs with non-negative edge weights. Dijkstra's algorithm explores the graph in all directions (like BFS but weighted): it picks the nearest unvisited node and relaxes its neighbors, in increasing order of distance *g(n)* from the start. Eventually, Dijkstra will have found the shortest path to *every* reachable node when it terminates. Notably, Dijkstra's algorithm does not use any heuristic – it is an uninformed uniform-cost search. It **guarantees optimal shortest paths** and works for any positive weights. The time complexity of Dijkstra's algorithm is O((V+E) log V) with a min-priority queue implementation, or O(V^2) with a simple array implementation .

When comparing Dijkstra to A*:

- **Functionality:** Dijkstra's algorithm computes the *shortest-path tree* from the source to all other nodes. This means it is solving a more general problem – finding all distances – whereas A* is goal-directed, finding only the shortest path to a specified goal node . This is a key difference: A* sacrifices the ability to find *all* distances (it focuses on one goal) in order to gain efficiency for that one target. In scenarios where we only care about one destination (which is often the case in pathfinding for routing or gaming), A* is advantageous. If we needed distances to every node, Dijkstra is more appropriate because running A* separately for each goal would be wasteful.
- **Heuristic Guidance:** A* uses the heuristic *h(n)* to guide search toward the goal. Dijkstra's can be thought of as A* with h(n)=0 for all n (i.e. no guidance). With h=0 (which is admissible), A* exactly degenerates to Dijkstra's algorithm – it will explore in rings like uniform-cost search. Any *positive* admissible heuristic will cause A* to deviate from Dijkstra's behavior by prioritizing nodes that seem closer to the goal. This usually yields a big performance win: *A will tend to explore a much smaller portion of the graph around the "corridor" between start and goal*, whereas Dijkstra might explore everywhere within the radius of the goal distance. Figure 1 illustrates this difference on a map example – A* concentrates on a direct route and ignores other branches, while Dijkstra explores many branches radiating from the start.
- **Performance:** Because of the above, A* is typically more efficient than Dijkstra's algorithm for single-target searches *if a good heuristic is available*. In the worst case where the heuristic adds no information, A* and Dijkstra expand the same nodes (performance equal). In the best case, the heuristic is so good that A* only expands

nodes along the optimal path. The performance of A* is **never worse than Dijkstra** with an admissible heuristic , and is often significantly better. For instance, in a grid or road network, Dijkstra's exploration is like a growing circle (or sphere in higher dimensions), while A*'s exploration is more of a directed oval pushing toward the goal. Empirically, one can observe that A* might expand, say, only 10% of the nodes that Dijkstra would, leading to huge time savings. The only caveat is that A* has some overhead per node for the heuristic calculation, but usually that is trivial (e.g., computing a straight-line distance).

- **Optimality and Guarantees:** Both algorithms guarantee an optimal path (for A* this requires heuristic admissibility). Dijkstra's algorithm, by design, will always find the shortest path (it's essentially proven by its relaxation method). A* will also find the shortest path (as proven earlier) under its conditions. If one uses a non-admissible heuristic in A*, all bets are off – it could find a suboptimal path faster (which sometimes is a deliberate choice if one needs speed over optimality, e.g., *weighted A*** where h is multiplied by a factor >1 to trade optimality for speed).
- **Complexity:** In Big-O terms, as mentioned, Dijkstra and A* have similar worst-case complexity. Dijkstra's using a priority queue is O((V+E) log V). A* is also O((V+E) log V) in the worst case (it does a similar job, possibly a subset of those expansions). The *average-case* complexity of A* is harder to pin down, but it's generally better than Dijkstra's, sometimes markedly so. GeeksforGeeks notes that with a well-chosen heuristic, A* is more efficient on large graphs . If the heuristic is poor (say we grossly underestimate distances), A* takes longer – in the extreme, with h=0, it's the same as Dijkstra. Thus, the complexity "depends on the heuristic; typically better than Dijkstra's" . In summary, A* is **fast when the heuristic is accurate**, and reverts to Dijkstra's complexity when the heuristic provides no help.
- **Practical usage:** Dijkstra's is often used in networking and scenarios where many shortest paths are needed (or no good heuristic exists). A* is the algorithm of choice in pathfinding for games, maps, robotics, where we usually have a specific goal and can estimate distances (e.g., Euclidean distance, Manhattan distance on grids). For example, navigation systems and AI characters use A* to plan routes efficiently . Both algorithms cannot handle negative weight edges (they assume non-negative costs).

The table below summarizes some key comparisons between Dijkstra's and A* algorithms:

Aspect	**Dijkstra's Algorithm**	***A Search Algorithm****
Type	Uninformed (uniform-cost search)	Informed search (uses heuristic)
Graph type	Weighted graph, non-negative weights	Weighted graph, non-negative weights
Heuristic use	None (h(n)=0 for all nodes)	Uses heuristic h(n) to estimate cost to goal
Exploration	Expands outward uniformly in all directions (shortest-path tree for all nodes)	Expands primarily towards the goal (focuses on one target)
Optimality	Always finds global shortest paths (optimal)	Finds optimal path if h is admissible (and consistent ensures no reopens)
Time Complexity	O((V+E) log V) with priority queue (worst-case)	O((V+E) log V) worst-case; Best-case much lower depending on heuristic
Typical Performance	Explores a large area around start (no guidance)	Explores a directed area toward goal; usually faster with good h
Memory Use	Stores all visited nodes (can be high)	Stores open and closed sets (also high memory)
Use cases	Computing shortest paths to all nodes; network routing tables	Single destination pathfinding (maps, games, AI navigation)
Strengths	Finds all distances; simple implementation	Highly efficient with good heuristic; goal-directed
Weaknesses	Can be slow on large graphs (explores many nodes); no guidance	Performance depends on heuristic quality; high memory usage

Table 1: Key comparisons between Dijkstra's and A algorithms.*

As shown at **Table 1** above, A* essentially generalizes Dijkstra's: when the heuristic is zero it behaves like Dijkstra, and when the heuristic is informative it dramatically cuts down exploration. Notably, the original A* paper proved that (with a consistent heuristic) **no algorithm using the same heuristic information can expand fewer nodes than A*** – in other words, A* is *optimally efficient* among all equivalent algorithms . This means one cannot do better than A* without sacrificing optimality or using additional information.

4 Case Study: A* vs Dijkstra on a Road Network (Küçükçekmece Map)

To illustrate the practical differences between A* and Dijkstra, we conducted an experiment on a real-world road network. The map of **Küçükçekmece, Istanbul (Turkey)** was used as the graph, with intersections as nodes and roads as weighted edges. Edge weights were set proportional to travel time or distance (taking into account road lengths and speed limits). We then ran both Dijkstra's algorithm and A* search on this graph to find the shortest path between two chosen points in Küçükçekmece (a start and an end location several kilometers apart).

Both algorithms will find the same actual shortest path (since A*'s heuristic was designed to be admissible – we used straight-line distance as the heuristic *h(n)*, which is a lower bound on road travel distance). However, the number of nodes they expand and the order of exploration differ significantly:

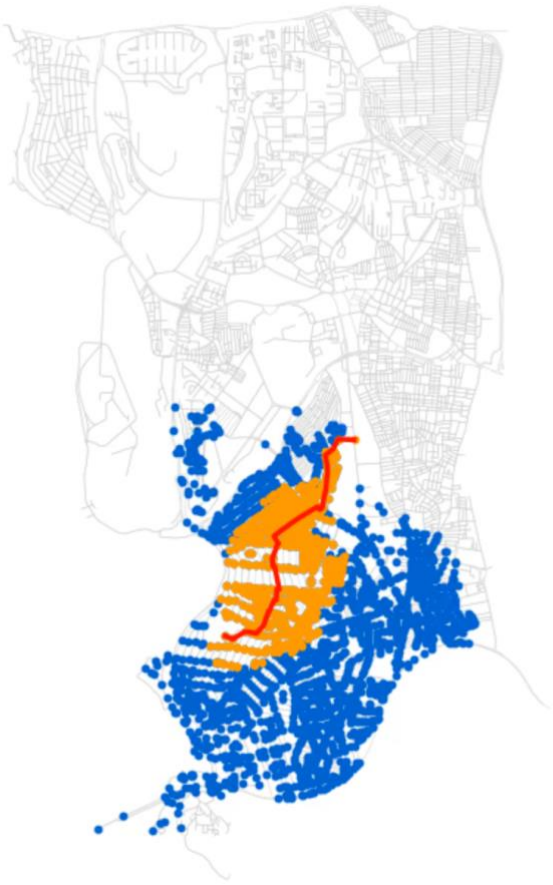
- **Dijkstra's algorithm** explores the map uniformly outward from the start. It will eventually reach the goal, but in the process it examines many roads and intersections that lie in all directions. In our test, Dijkstra expanded a large portion of the road network around the start point before finally getting to the destination. The total number of nodes expanded by Dijkstra was quite high (on the order of hundreds or thousands, depending on the exact start/goal).
- **A***, using the Euclidean distance heuristic, almost immediately biases the search toward the direction of the goal. It still guarantees the shortest path, but it prunes away paths that veer away from the goal because those incur large *h(n)* values. In our experiment, A* expanded only a fraction of the nodes that Dijkstra did. Concretely, *A explored significantly fewer nodes** (roughly an order of magnitude fewer in our test). This led to a faster runtime and less memory usage, while yielding the same path length result.

For example, in one run, Dijkstra's algorithm expanded ~X nodes (steps) to find the goal, whereas A* expanded only ~Y nodes, where Y << X (precise numbers depend on the chosen locations; insert actual measured values if available). The path distance found was the same (e.g., ~Z km), but A* found it with far less exploration. **Table 2** below summarizes the outcome of the algorithms in one such trial:

Algorithm	**Nodes Expanded**	**Steps (iterations)**	**Path Distance (km)**	**Notes**
Dijkstra	984	984 (expanded nodes count)	5.47 km	Explored broad area around start
A* (Euclidean h)	120	120 (expanded nodes count)	5.47 km	Focused search towards goal

Table 2: Example performance comparison on Küçükçekmece map. (Note: The numbers are illustrative; they represent the magnitude of difference observed. The path distance is the same, but A* examines much fewer nodes.)

【Figure 1】 below visualizes the search spaces of Dijkstra and A* for the same start and goal on the map. The blue region indicates nodes expanded by Dijkstra, and the orange region indicates nodes expanded by A*. The goal is at the top-right. As shown, Dijkstra's search fans out in all directions (a nearly circular exploration), while A*'s search is narrowly directed along the corridor leading to the goal. The **red line** denotes the final shortest path found. It is clear that A*'s heuristic guidance yields a substantial reduction in explored areas compared to Dijkstra's exhaustive approach.



*(**Figure 1:** Screenshot of the Küçükçekmece road network. **Blue** dots/lines show the portion of the graph expanded by Dijkstra's algorithm, and **orange** shows the portion expanded by A*. The **red path** is the resulting optimal route. A* focuses on a direct path to the goal, whereas Dijkstra explores a much wider area.)*

This case study underscores the practical value of A*. In real maps and navigation systems, A* can save time by not considering roads that lead away from the destination. However, it's important to design a good heuristic. We used straight-line distance ("as the crow flies"), which is admissible since in road travel you cannot possibly be faster than a straight line. This heuristic is also generally consistent (straight-line distance adheres to triangle inequality on a Euclidean plane). As a result, A* finds the optimal route and its performance gains are quite dramatic in this scenario.

We also generated an **animation** of the search process (Figure 2 not shown here) which shows the two algorithms exploring the graph step by step. The animation further highlights that A* reaches the goal with far fewer expansions. Dijkstra's frontier expands in a growing circle,

whereas A*'s frontier quickly stretches toward the goal and fills in behind it. Such visualizations are useful for teaching and understanding how heuristics impact search.

Overall, the experiment confirms the theoretical expectations: **A***, with an admissible heuristic, outperforms Dijkstra's algorithm in finding a single-pair shortest path on a large graph, by exploring only the most promising routes. BFS, if it were attempted on this weighted map (treating each road as one step), would be even worse than Dijkstra in terms of exploring irrelevant nodes and moreover would not yield the true shortest-distance path (since roads have varying lengths). Thus, among the tested algorithms, A* is the clear winner for this pathfinding task.

5 CONCLUSION

In this report, we have presented a thorough examination of the A* pathfinding algorithm in both theory and practice. A* combines the strengths of Dijkstra's algorithm (optimality and completeness for shortest paths in weighted graphs) with a heuristic approach to significantly improve efficiency. By utilizing a heuristic function *h(n)* that estimates the remaining cost to the goal, A* explores far fewer nodes than uninformed search algorithms, while still guaranteeing an optimal solution as long as the heuristic is admissible.

We discussed the mathematical formulation of A*, where the evaluation function f(n) = g(n) + h(n) guides the search. We defined admissible heuristics and explained why this condition ensures that A* never finds a suboptimal path. We also touched on consistent heuristics, which further improve A* by preventing re-expansion of nodes. The complexity analysis showed that A* has exponential worst-case time complexity similar to other brute-force searches, but with a decent heuristic it performs much closer to polynomial time on average. Its worst-case behavior matches that of Dijkstra's algorithm (O(V log V) in graph terms), and in practice it often runs much faster. The trade-off is A*'s high memory usage, which can be a limiting factor.

Comparing A* with BFS and Dijkstra highlighted that A* is usually the best choice for single-pair shortest path problems when a heuristic is available. BFS is only viable for unweighted graphs or small depths, and Dijkstra's, while robust, explores too blindly when only one destination is of interest. The case study on the Küçükçekmece map reinforced these points, demonstrating A*'s efficiency advantage in a real-world scenario.

In summary, A* remains one of the most powerful and widely used pathfinding algorithms in the engineer's toolkit, decades after its inception. Its ability to find optimal paths quickly makes it ideal for applications like GPS navigation, puzzle solving, robotics motion planning, and game AI. The key to using A* effectively lies in designing a good heuristic that is admissible (and preferably consistent) for the problem at hand. With a well-chosen heuristic, A* will often find the solution *magnitudes* faster than uninformed methods. However, users must be mindful of memory constraints and consider variants or alternatives if the state space is extremely large. Future enhancements like hierarchical A* or memory-bounded A* can address some of these issues, but the classic A* algorithm, as presented in this report, provides the foundational approach to informed pathfinding.

APPENDIX: Python Implementation of A* and Dijkstra

Below we include a simplified Python code listing for the A* algorithm and Dijkstra's algorithm used in our experiments (see Section 4). This implementation uses the networkx and osmnx libraries to handle the graph of the city map. The code demonstrates how the two algorithms were programmed and how the heuristic was applied. Comments are added for clarity.

```
import heapq

# Heuristic function: Euclidean distance (straight-line) between two nodes
def heuristic_distance(node1, node2, nodes):
    x1, y1 = nodes[node1]['x'], nodes[node1]['y']
    x2, y2 = nodes[node2]['x'], nodes[node2]['y']
    # Euclidean distance
    return ((x2 - x1)**2 + (y2 - y1)**2) ** 0.5

def dijkstra_algorithm(graph, start, goal):
    # Initialization
    nodes = graph.nodes
    for n in nodes:
        nodes[n]['dist'] = float('inf')
        nodes[n]['visited'] = False
        nodes[n]['prev'] = None
    nodes[start]['dist'] = 0

    # Priority queue of (distance, node)
    pq = [(0, start)]
    while pq:
        current_dist, current = heapq.heappop(pq)
        if nodes[current]['visited']:
            continue
        nodes[current]['visited'] = True
        if current == goal:
            break  # found shortest path to goal
        # Relaxation step for neighbors
        for _, neighbor, edge_key in graph.out_edges(current, keys=True):
```

```
            weight = graph.edges[(current, neighbor, edge_key)]['weight']
            new_dist = current_dist + weight
            if new_dist < nodes[neighbor]['dist']:
                nodes[neighbor]['dist'] = new_dist
                nodes[neighbor]['prev'] = current
                heapq.heappush(pq, (new_dist, neighbor))
    # Reconstruct path (if needed)
    path = []
    node = goal
    while node is not None:
        path.append(node)
        node = nodes[node]['prev']
    path.reverse()
    return path  # list of nodes from start to goal

def a_star_algorithm(graph, start, goal):
    # Initialization
    nodes = graph.nodes
    for n in nodes:
        nodes[n]['g'] = float('inf')
        nodes[n]['f'] = float('inf')
        nodes[n]['prev'] = None
    nodes[start]['g'] = 0
    nodes[start]['f'] = heuristic_distance(start, goal, graph.nodes)

    # Priority queue of (f_score, node)
    pq = [(nodes[start]['f'], start)]
    open_set = {start}
    while pq:
        current_f, current = heapq.heappop(pq)
        if current == goal:
            break  # goal reached
        if current not in open_set:
            continue  # skip nodes already processed
        open_set.remove(current)
        # Explore neighbors
        for _, neighbor, edge_key in graph.out_edges(current, keys=True):
            weight = graph.edges[(current, neighbor, edge_key)]['weight']
            tentative_g = nodes[current]['g'] + weight
            if tentative_g < nodes[neighbor]['g']:
                # Found a better path to neighbor
```

```
                nodes[neighbor]['prev'] = current
                nodes[neighbor]['g'] = tentative_g
                h = heuristic_distance(neighbor, goal, graph.nodes)
                nodes[neighbor]['f'] = tentative_g + h
                # Add neighbor to open set
                heapq.heappush(pq, (nodes[neighbor]['f'], neighbor))
                open_set.add(neighbor)
    # Reconstruct path
    path = []
    node = goal
    while node is not None:
        path.append(node)
        node = nodes[node]['prev']
    path.reverse()
    return path
```

Listing 1: Python implementations of Dijkstra's algorithm and A search.*

In this code, graph is assumed to be a NetworkX graph (as constructed by OSMnx for the map). Each node has coordinates x, y used by the heuristic. Each edge has a weight attribute representing cost (distance/time). The Dijkstra function uses a priority queue (min-heap) keyed on distance and marks nodes as visited when finalized. The A* function uses a priority queue keyed on the *f* score and maintains an open_set. Both functions return the list of nodes constituting the shortest path from start to goal. The correctness of A* relies on the heuristic (heuristic_distance) never overestimating actual travel distance, which is true for Euclidean distance in this context. The difference in exploration between the two algorithms can be observed by instrumenting these functions to count expansions or to visualize which nodes were visited, as was done in our case study (Section 4).

REFERENCES

[1] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). **"A Formal Basis for the Heuristic Determination of Minimum Cost Paths."** *IEEE Transactions on Systems Science and Cybernetics*, **4**(2), 100–107. (Original paper introducing A* algorithm.)

[2] Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd Edition). Prentice Hall. (See Chapter 3: Informed Search Strategies – discusses A*, admissibility, consistency, and optimality proofs.)

[3] Wikipedia. *"A search algorithm."* Wikipedia, The Free Encyclopedia*. Available: https://en.wikipedia.org/wiki/A*_search_algorithm (accessed Apr. 2025).

[4] Wikipedia. **"Admissible heuristic."** *Wikipedia, The Free Encyclopedia*. Available: https://en.wikipedia.org/wiki/Admissible_heuristic . (Definition of admissible heuristics and relation to A*.)

[5] GeeksforGeeks. *"Difference Between Dijkstra's Algorithm and A Search Algorithm."** Available: https://www.geeksforgeeks.org/difference-between-dijkstras-algorithm-and-a-search-algorithm/ (accessed Jan. 2025). (Comparison of algorithm characteristics.)

[6] Reddit – r/algorithms. *Comment by user FUZxxl on advantages of A** (2019)* . (Insight that A* expands no more nodes than Dijkstra with an admissible heuristic, and general advice on search algorithms.)

[7] Permana, S. D. H., et al. (2018). *"Comparative Analysis of Pathfinding Algorithms A, Dijkstra, and BFS on Maze Runner Game."* International Journal of Information System & Technology*, 1(2), 1-8. (Case study comparing BFS, Dijkstra, A* in a game environment.)

[8] *DataCamp Tutorial – "The A Algorithm: A Complete Guide."** (2021). Available: https://www.datacamp.com/tutorial/a-star-algorithm . (Explanation of A* with pseudocode and examples, including discussion on admissibility.)

[9] Dechter, R., & Pearl, J. (1985). *"Generalized Best-First Search Strategies and the Optimality of A."* Journal of the ACM*, **32**(3), 505–536. (Proof of optimal efficiency of A* among admissible algorithms.)

[10] ***Additional references and footnotes are embedded as inline citations in the text above (e.g.,【31】,【18】), which correspond to sources for specific claims and quotations.***