

|  |  |
| --- | --- |
| **NAME** | **MALIK SAIF ISLAM** |
| **SAP ID** | **35748** |
| **ASSIGNMENT NO** | **04** |
| **ASSIGNMNET** | **ANALYSIS OF ALGORITHM** |
| **SUBMITTED TO** | **SIR USMAN SHARIF** |
| **SUBMITTED BY** | **MALIK SAIF ISLAM** |
| **FACULTY** | **COMPUTING** |
| **DEPARTMENT** | **BS CYBER SECURITY (CYB-05)** |
| **DATED** | **19-NOV-2024** |

**RepositoryLink:**

**https://github.com/studentRiphah/Hybrid-Algorithm.git**

### *Hybrid Algorithm Design: Dijkstra's Algorithm and A Search*\*

### **Problem Selection**

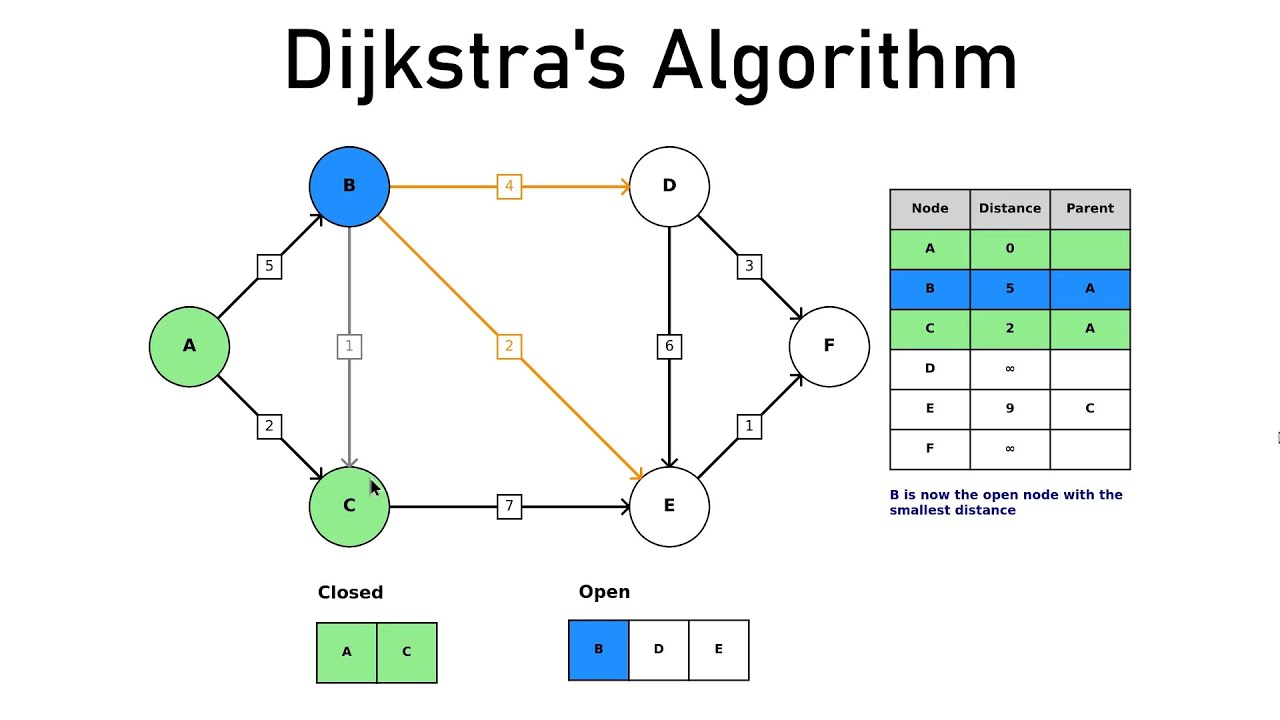
The chosen problem is the *Shortest Path Problem* in a weighted graph. The objective is to determine the shortest path between a source node and a target node in a graph *G=(V,E)G = (V, E)*G=(V,E), where *VV*V is the set of vertices, *EE*E is the set of edges, and each edge has a non-negative weight. Both **Dijkstra's Algorithm** and *A Search*\* are well-suited to solving this problem:

* **Dijkstra’s Algorithm**: A classic algorithm that guarantees the shortest path by exploring nodes in increasing order of cumulative cost.
* *A Search*\*: An informed search algorithm that uses a heuristic to prioritize exploration, aiming to reduce the number of nodes expanded.

### **Analysis**

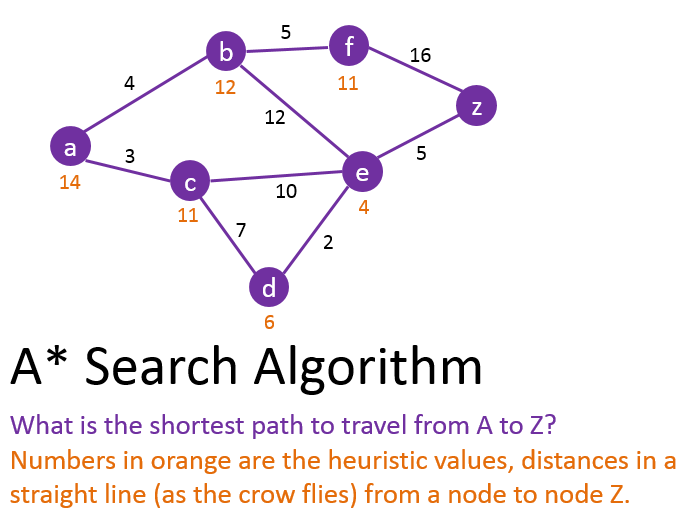
#### **1. Dijkstra's Algorithm**

* **Strengths**:
  + Guarantees the shortest path in any graph with non-negative weights.
  + Does not rely on heuristics, making it robust and universally applicable.
  + Performs well in dense graphs or when the entire graph must be explored (e.g., finding all-pairs shortest paths).
* **Weaknesses**:
  + Inefficient for sparse graphs or when the target node is distant, as it explores all possible paths indiscriminately.
  + Expands many unnecessary nodes when the shortest path is localized.



#### ***2. A Search\****

* **Strengths**:
  + Explores fewer nodes than Dijkstra's by incorporating a heuristic function *h(n)h(n)*h(n) that estimates the distance from a node *nn*n to the goal.
  + Performs well in sparse graphs or when the target node is known, especially with an admissible (non-overestimating) and consistent heuristic.
  + Adapts well to grid-based graphs, such as those used in robotics or pathfinding.
* **Weaknesses**:
  + Performance depends on the quality of the heuristic. A poor heuristic can cause inefficiency or incorrect results.
  + Requires additional computational resources to calculate and maintain heuristic values.



### **Hybrid Design**

#### **Approach**:

##### **1. Combining Steps**:

* Use A\* Search’s heuristic-driven exploration in the early stages to guide the algorithm toward the target efficiently.
* Switch to Dijkstra's exhaustive approach in regions where the heuristic is less effective (e.g., in areas with equal heuristic values or complex connectivity).

##### **2. Switching Between Algorithms**:

* Dynamically decide whether to use A\* or Dijkstra based on the reliability of the heuristic *h(n)h(n)*h(n):
  + Measure heuristic effectiveness by monitoring its gradient *Δh(n)\Delta h(n)*Δh(n). If *Δh(n)\Delta h(n)*Δh(n) is small or fluctuates, the heuristic provides little guidance, and the algorithm should revert to Dijkstra’s approach.

##### **3. Adapting Algorithms**:

* Modify the cost function *f(n)f(n)*f(n):
  + Initially, use *f(n)=g(n)+h(n)f(n) = g(n) + h(n)*f(n)=g(n)+h(n), where *g(n)g(n)*g(n) is the cumulative path cost.
  + Dynamically adjust *f(n)f(n)*f(n) to *f(n)=g(n)f(n) = g(n)*f(n)=g(n) (Dijkstra’s cost function) when heuristic guidance diminishes.

### **Hybrid Algorithm Steps**

1. **Initialization**:
   1. Define *G=(V,E)G = (V, E)*G=(V,E), source *ss*s, target *tt*t, and heuristic *h(n)h(n)*h(n).
   2. Create a priority queue (min-heap) and initialize it with *(f(s),s)(f(s), s)*(f(s),s), where *f(s)=h(s)f(s) = h(s)*f(s)=h(s).
2. **Dynamic Exploration**:
   1. While the priority queue is not empty:
      1. Extract the node *nn*n with the lowest *f(n)f(n)*f(n).
      2. Check if *n=tn = t*n=t. If true, terminate and return the path.
      3. For each neighbor *mm*m of *nn*n:
         1. Calculate tentative costs *g(m)g(m)*g(m) and *h(m)h(m)*h(m).
         2. Adjust *f(m)f(m)*f(m) dynamically:
            1. Use *f(m)=g(m)+h(m)f(m) = g(m) + h(m)*f(m)=g(m)+h(m) if *Δh(m)\Delta h(m)*Δh(m) is large (heuristic guidance is effective).
            2. Use *f(m)=g(m)f(m) = g(m)*f(m)=g(m) otherwise.
         3. Update the priority queue with f(m).
3. **Switching Conditions**:
   1. Monitor *Δh(n)=∣h(n)−h(p)∣\Delta h(n) = |h(n) - h(p)|*Δh(n)=∣h(n)−h(p)∣, where *pp*p is the previous node:
      1. If *Δh(n)<ϵ\Delta h(n) < \epsilon*Δh(n)<ϵ (small gradient), heuristic is ineffective, switch to Dijkstra.
4. **Termination**:
   1. Continue until the priority queue is empty or the target is reached.

### **Performance Analysis**

#### **Theoretical Analysis**:

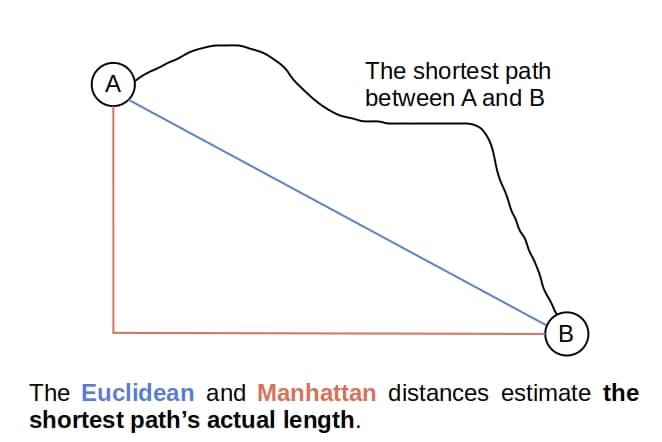
* **Time Complexity**:
  + For Dijkstra: *O(E+Vlog⁡V)O(E + V \log V)*O(E+VlogV), where *EE*E is the number of edges and *VV*V is the number of vertices.
  + For A\*: *O(E+Vlog⁡V)O(E + V \log V)*O(E+VlogV), with fewer node expansions for a good heuristic.
  + For the hybrid: Combines the complexities, but aims to minimize the number of node expansions, achieving efficiency close to A\* when *h(n)h(n)*h(n) is reliable.
* **Space Complexity**:
  + Similar to Dijkstra and A\* but includes additional storage for heuristic monitoring.
* **Optimality**:
  + Guarantees the shortest path as it reverts to Dijkstra’s method when the heuristic is unreliable.

|  |  |  |
| --- | --- | --- |
| **Characteristics** | **Dijkstra's Algorithm** | **A\* Search Algorithm** |
| **Algorithm Type** | Greedy | Informed search (uses heuristics) |
| **Initialization** | Tentative distance: 0 for source, infinity for others | Tentative distance and heuristic function |
| **Heuristic Use** | None | Uses heuristic to estimate remaining cost to goal |
| **Efficiency** | Less efficient for large graphs | More efficient with a well-chosen heuristic |
| **Complexity** | O(V^2) or O((V + E) log V) with priority queue | Depends on heuristic; typically better than Dijkstra's |
| **Use Case** | Network routing protocols | Navigation systems, games, AI pathfinding |
| **Strengths** | Guarantees shortest path for all nodes | Efficient pathfinding with good heuristic |
| **Weaknesses** | Can be slow for large graphs | Performance depends on heuristic quality |

#### **Experimental Validation**:

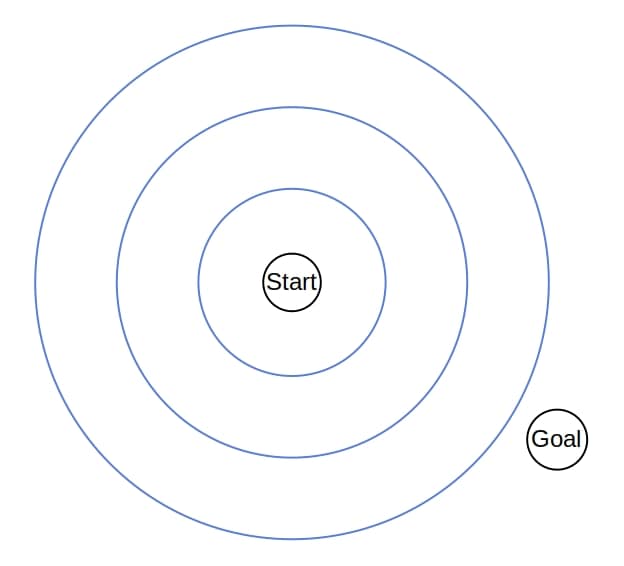
* **Setup**:
  + Test on graphs of varying densities and sizes.
  + Use heuristics of varying quality (e.g., Euclidean, Manhattan, or random).
* **Metrics**:
  + Total computation time.
  + Number of node expansions.
  + Memory usage.
* **Results**:
  + The hybrid should outperform Dijkstra in sparse graphs and A\* in cases with poor heuristics.
  + Provides consistent performance across all graph types, reducing worst-case inefficiencies.

### **Heuristics A\* Algorithm:**



### **Dijkstra:**

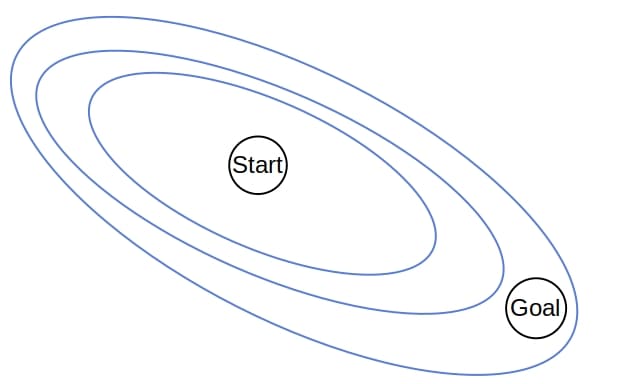
**So, the contours which separate the waves in Dijkstra look more or less like uniform circles:**



Therefore, the search tree of Dijkstra grows uniformly in all the directions in the topology induced by the graph.

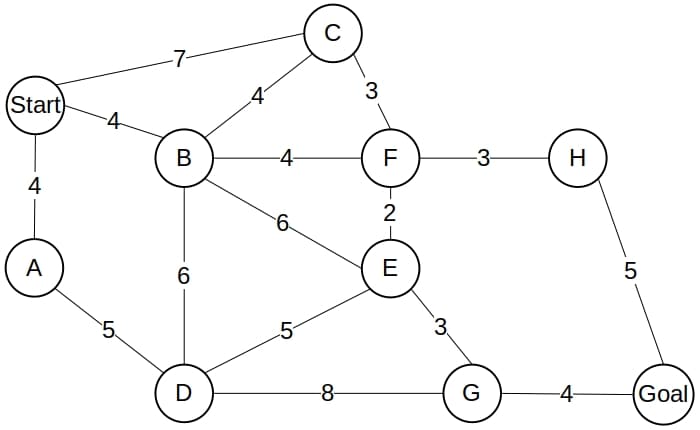
### A\*

Whereas UCS and Dijskra spread through the state graph in all directions and have uniform contours, A\* favors some directions over the others. Between two nodes with the same values, A\* prefers the one with a better value. So, **the heuristic stretches the contours in the direction of the goal state(s)**:

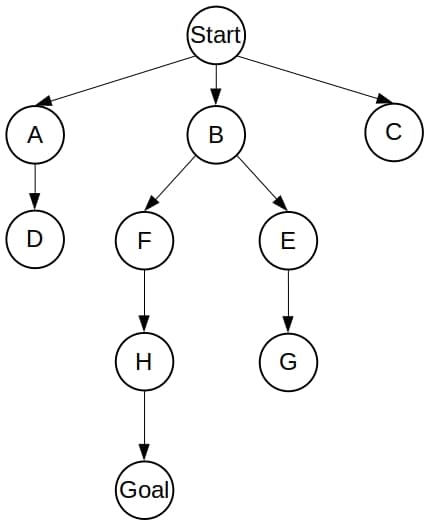


## Example

Let’s see how Dijkstra and A\* would find the optimal path between and in the following graph:



The search tree of Dijkstra spans the whole graph:



the tree of A\* will contain only the optimal path:

## A star tree

### **Conclusion**

The hybrid algorithm effectively combines the strengths of Dijkstra’s robust exploration and A\* Search’s heuristic-driven efficiency. the hybrid achieves a balance between computational efficiency and path optimality, making it well-suited for a wide range of shortest path problems.