Эвристический алгоритм — это алгоритм решения задачи, правильность которого для всех возможных случаев не доказана, но про который известно, что он даёт достаточно хорошее решение в большинстве случаев. В действительности может быть даже известно (то есть доказано) то, что эвристический алгоритм формально неверен. Его всё равно можно применять, если при этом он даёт неверный результат только в отдельных, достаточно редких и хорошо выделяемых случаях, или же даёт неточный, но всё же приемлемый результат.
Проще говоря, эвристика — это не полностью математически обоснованный (или даже «не совсем корректный»), но при этом практически полезный алгоритм.

Важно понимать, что эвристика, в отличие от корректного алгоритма решения задачи, обладает следующими особенностями:

- Она не гарантирует нахождение лучшего решения.
- Она не гарантирует нахождение решения, даже если оно заведомо существует (возможен «пропуск цели»).
- Она может дать неверное решение в некоторых случаях.

Алгоритм поиска A* (произносится: «А звездочка») — относится к эвристическим алгоритмам поиска по первому лучшему совпадению на графе с положительными весами ребер, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).
Алгоритм был описан в 1968 году Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем.

Алгоритм использует вспомогательную функцию (эвристика), чтобы направлять поиск решения и сокращать его длительность. Алгоритм полный в том смысле, что всегда находит оптимальное решение, если он существует.

Идея алгоритма. A* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдет минимальный путь. Как и все эвристические алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От «**жадного алгоритма**», который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая g(x) — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение f(x), после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетами. Приоритет пути определяется по значению f(x) = g(x) + h(x). Алгоритм продолжает свою работу до тех пор, пока значение f(x) целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

Для расчета реализуем две функции:
FUNCTION A*(start,end) и
FUNCTION RECONSTRUCT_PATH(fromset,start,end)

Псевдокод:

```
FUNCTION A*(start,end)
     closedset = the empty set  // Множество вершин, которые уже были обработаны(раскрыты)
     openset = {start}          // Множество вершин(очередь), которые предстоит обработать(раскрыть).
                                // Изначально здесь присутствует только начальная вершина start.
     fromset = the empty set    // Карта пройденных вершин. Используется функцией RECONSTRUCT_PATH
                                   для восстановления пути (вывода результата).
     //Заполняем свойства вершины start
     G(start) = COST(start,start) = 0  // Стоимость пути от начальной вершины. У start g(x) = 0.
     F(start) = G(start) + H(start,end) // h(x) - эвристическая оценка расстояния до цели.
     //Основной цикл алгоритма
     WHILE openset IS NOT EMPTY
       curr = MIN_F(openset)  // Вершина из openset имеющая самую низкую оценку f(x).
       IF (curr = end) RETURN  RECONSTRUCT_PATH(fromset,start,end) // Выводим результат.
       REMOVE curr FROM openset // Вершина curr пошла на обработку, а значит её следует удалить из
                                   очереди на обработку.
       ADD curr TO closedset    // И добавить в список уже обработанных.
       FOREACH neighbour OF curr neighbours // Проверяем каждого соседа curr
          IF neighbour IN closedset CONTINUE   // Пропускаем соседей из закрытого списка
(предварительный, ожидаемый) tentative_g_score = G(curr) + COST(curr,neighbour)  // Вычисляем g(x)
                                                                                   для обрабатываемого соседа
          IF neighbour NOT IN openset // Если сосед curr ещё не в открытом списке
              ADD neighbour TO openset //добавим его туда
              tentative_is_better = TRUE // вводим признак того, что нужно обновить свойства для
                                            соседней вершины
          ELSE         // Сосед был в открытом списке, а значит мы уже знаем его g(x), h(x) и f(x)
              IF tentative_g_score < G(neighbour) // Вычисленная g(x) оказалась меньше, а значит
                                                     нужно будет обновить значения g(x), h(x), f(x)
                 tentative_is_better = TRUE
              ELSE   // Вычисленная g(x) оказалась больше, чем имеющаяся в openset.
```

```
                    // Это означает, что из вершины curr путь через этого соседа дороже
                    // т.е. существует менее дорогой маршрут, пролегающий через этого соседа (из
                         какой-то другой вершины, не из curr)
                    // Поэтому данного соседа мы игнорируем
                    tentative_is_better = FALSE
          // Обновление свойств соседа.
          IF tentative_is_better = TRUE
              fromset(neighbour) = curr //Вершина с которой мы пришли.
                                          Используется для реконструкции пути.
              G(neighbour) = tentative_g_score
              F(neighbour) = G(neighbour) + H(neighbour, end)
        // Обратите внимание, что если происходит обновление свойств - значит neighbour(сосед curr)
             так или иначе находится в openset.
        // Т.е. при следующей итерации внешнего цикла из openset будет
             извлечена вершина с наименьшей оценкой f(x).
        // Не исключено, что она окажется соседом нашего curr, которого мы только что добавили.
        // В общем это самая важная особенность алгоритма A*.
RETURN FAILURE //управление передаётся сюда когда openset пуст,
                  а вершина end не найдена (путь найти не удалось)

// Восстанавливаем результироующий путь.
// Путь можно проследить только от заданной вершины(чаще всего это end)
     к старту(каждая вершина имеет пути находится в fromset, чем мы и воспользуемся).
FUNCTION RECONSTRUCT_PATH(fromset,start,end)
   pathset = the empty list // Упорядоченное множество результирующих вершин пути.
     curr = end // Поиск начинается от финиша.
     ADD curr TO pathset  // Добавляем end в результирующий путь.
     WHILE curr <> start  // Добавляем в путь все вершины от end до start.
          curr = fromset(curr) // Получаем вершину из которой пришли в curr.
          ADD curr TO pathset  // Добавить вершину в результирующий путь.
RETURN REVERSE(pathset) // Так как мы построили путь от end к start, то результирующий набор вершин
                           необходимо перевернуть.
```

Эвристические функции. Поведение алгоритма сильно зависит от того, какая эвристика используется. В свою очередь, выбор эвристики зависит от постановки задачи. Часто A* используется для моделирования перемещения по поверхности, покрытой координатной сеткой.

Если мы можем перемещаться в четырех направлениях, то в качестве эвристики стоит выбрать **манхэттенское расстояние** (расстояние городских кварталов, метрика прямоугольного города) — метрика, введённая Германом Минковским. Согласно этой метрике, расстояние между двумя точками равно сумме модулей разностей их координат:

$$h(u) = |u.x - goal.x| + |u.y - goal.y|.$$

Расстояние Чебышева применяется, когда к четырем направлениям добавляются диагонали:

$$h(u) = \max(|u.x - goal.x|,|u.y - goal.y|).$$

Если передвижение не ограничено сеткой, то можно использовать **евклидово расстояние** по прямой:

$$h(u) = sqrt((u.x - goal.x)^2 + (u.y - goal.y)^2).$$

Также стоит обратить внимание на то как соотносятся g(x) и h(x). Если они измеряются в разных величинах (например, g(x)— это расстояние в километрах, а h(x) — оценка времени пути в часах) A* может выдать некорректный результат.

Свойства. Алгоритм A* обладает следующими свойствами:

- *допустимости:* если решение существует, он будет найден.

- *оптимальность:* найденное решение всегда оптимален. Если решений несколько, будет найден один из них (в зависимости от деталей реализации алгоритма).

- *эффективность:* не существует алгоритмов, которые находят решение быстрее с применением той же эвристической функции (точнее: *A* раскрывает минимальное количество вершин.).*

Временная сложность. Обычно алгоритм A* просматривает только часть вершин. Однако, в лабиринтах быстродействие приближается к худшему случае. Быстродействие алгоритма существенно зависит от двух факторов:

-*Точность эвристической функции*.

-*Реализация контейнеров известных и исследованных вершин:* Наиболее затратными операциями в алгоритме есть операции сложения, изъятия и изменения элементов в списках известных и исследованных вершин. По их быстродействие существенно влияют конкретные реализации этих структур данных.

Пусть имеется множество вершин в графе, информация о вершинах и ребрах доступна до начала работы алгоритма, использованная эвристическая функция — монотонная. Список известных вершин реализован как бинарная куча, список исследованных — как массив. Тогда алгоритм A* имеет квадратичную зависимость от количества вершин графа и худшее время работы:

$$O(|V|^2)$$

Функция **MIN_F(openset)** может быть оптимизирована. Если каждая вершина хранится как указатель на соответствующий объект в куче, то время работы функции уменьшится с квадратичного до логарифмического, а общее время работы алгоритма — до линейно-логарифмического:

$$O(|V|*\log|V|)$$