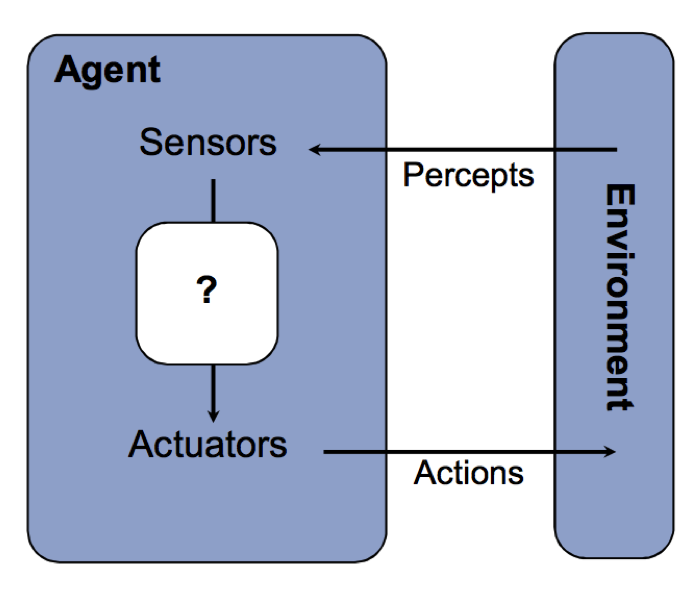
**What is an agent?**

An **agent** is any entity that can perceive its environment through sensors and acts on its environment through effectors.



If you think of humans as agent, we have eyes, ears, and other organs for sensing our world, and then hands, legs, etc that are effectors. What we are most interested in AI is what decisions are made, and how, given the data that comes in through the sensors. In the Agent image shown above, it's the box with the ? that we're going to be discussing all semester.

A rational agent use percepts of its environment to select rational actions, which is an action that maximizes its utility function.

In the video game [Pacman](https://www.youtube.com/watch?v=uswzriFIf_k" \t "_blank), the pacman is the agent with a few objectives, depending on who's playing the game:

1. Eat all of the dots quickly.
2. Get as many points as possible.
3. Don't die

Rational actions for Pacman include:

1. Move toward dots.
2. Move avoid ghosts
3. Move toward fruit.

When evaluating pacman as an agent, consider:

1. What is pacman's environment? *(dots, ghosts, fruit, walls)*
2. What does pacman perceive? (The pacman perceives what the person playing pacman perceives.) *(everything)*
3. What are pacman's actions? *(move in a direction, NSEW)*
4. How do pacman's actions update the environment? *(dots disappear, fruit disappears, ghosts change direction)*

**Reflex vs. goal-based agents**

Imagine you were writing a program to play pacman. As a pacman player, you understand the rules of the game and when playing, you observe the entire game board to see where the ghosts are and you take the shortest path to uneaten dots. You also go after fruit. You want to incorporate all of this knowledge into a computer program. There are a few things you need: a representation of the pacman world, defined goals for pacman, the ability to maintain the game state, and search algorithms to find the rational course of action for pacman.

**Reflex agents**

Reflex agents are the simplest agents and don't have all of the functionality just described. They choose an action based on the current perception of the world rather than planning or considering future consequences. For example, in pacman, there could be a rule such as "move away from ghosts", or "If ghost within two moves, move opposite direction". This is a good rule in the short term for staying alive, but could lead pacman to be trapped in a corner. For human agents, a reflex action is one that we take without thinking. If you touch a hot stove, you immediately move your hand away without considering any other dangers in doing so.

A reflex pacman agent will be far easier to implement than one that plans its actions using goals it has set. However, it's also limited in its capabilities and really not that interesting.

**Goal-based agents**

Goal-based agents plan ahead by considering "what if" a certain action is taken. What will the state of the environment be after the action and does it move the agent closer to its goal? To support goal-based agents, a model of the environment is needed that shows how the environment evolves in response to those actions. The model likely will not include the entire environment, but rather, just the elements of the environment that are needed to measure success. For example, if the goal in pacman is to eat dots, then there needs to be a measure of how many dots are in the environment. If the goal is to score points, then points needs to be measured. Knowing how many points are scored for a given action, as compared to other actions, provides pacman with information for choosing an action.

**Searching for a plan to achieve goals**

Whether we're talking about pacman, or something else, achieving goals requires having a plan. Your **plan** is the set of steps you're going to take to achieve that goal you've set. For pacman, the goal could be to get the dots quickly or to get the most points, and each of those goals requires a plan for the actions that pacman is going to take. Different goal = different plan.

Search algorithms are used to identify the best plan from the set of possible plans. You've all heard of algorithms such as breadth-first and depth-first search. In these algorithms, there is a starting state and a set of possible steps that can be taken from that starting state. The search algorithms evaluate the next states until the goal state is found.

To set up a problem as a search problem, it needs to be formulated in a search framework, which includes:

* State space: details of the environment that matter for achieving the goal
* Successor function: determines new state from current state given costs and actions
* Start state: current state of the environment
* Goal state: state of the environment we want to achieve
* Goal test: are we there yet? e.g. are all dots gone?

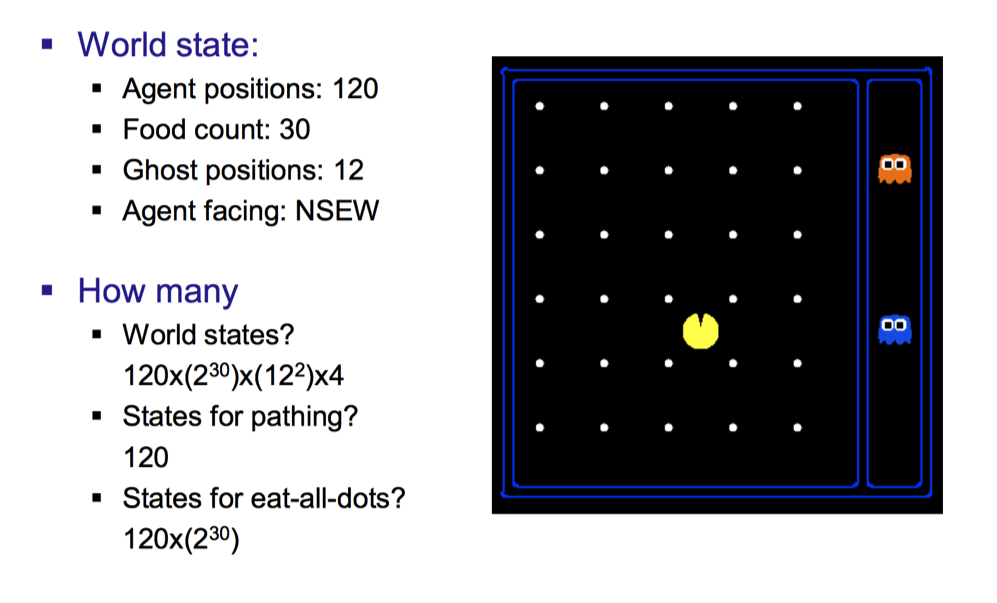
The search algorithm uses the formulation to find a solution to the problem. The solution is the sequence of actions, aka a plan, that transforms the start state into the goal state. Clearly, some plans are better than others. For example, pacman could have a goal to eat all dots by moving randomly about the game board. Or, pacman could have a plan to identify the remaining dots and move along the shortest path to eat the dots. A search algorithm would evaluate both plans and show that the targeted approach was better than the random approach.

**State space**

The state space includes all of the variables in the environment where the agent lives, and the combinations of values that those variables can have. A well-defined state space is required for creating goal-based agents to know whether a goal has been achieved. In any environment, there are typically elements in a state space that are not needed for measuring a goal and including them would generate way too many variable combinations to evaluate in a timely fashion.

There is a distinction between the World State, which includes all variables in the environment even if they're not needed in the search problem, and the Search State, which are only the details needed to solve the problem. When we refer to the world state space, we're referring to both the variables that are included in a world state and all combinations of those variables. In the pacman example shown here, there are four variables in the world state space and 120\*(230)\*(122)\*4 combinations of those variables. The size of the state space is 120\*(230)\*(122)\*4.

**World State**



In simple pacman example shown in image, the world state includes:

* Pacman's position
* Ghosts position
* Pellets remaining
* Pacman's direction (NSEW)

**Search state**

The search state uses only the relevant information from the world state needed to solve the problem. No other information about the environment is included, which reduces the search space.

**Example: Find a path for pacman from current location to a destination location**

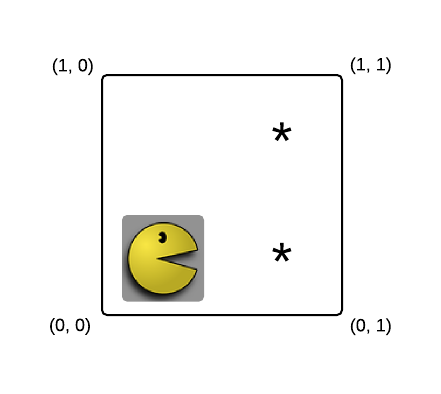
**State:** (x,y) location

**Next state:** current location + step in NSEW direction, set in a successor function

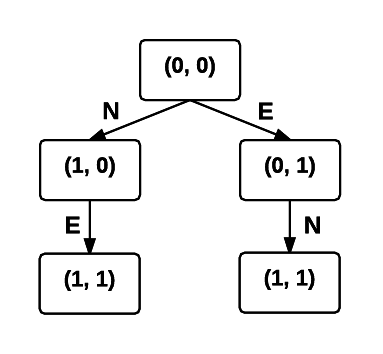
**Goal test:** (x,y) location = END

In this example, only the location is needed. The next state is determined in a successor function that uses the current location and the direction that pacman is facing and taking one step in that direction.

Consider a start state that looks like the following:



Pacman is at (0, 0). Assume you want to find a path for pacman to (1,1). The start state is (0, 0) and a successor function that calculates the next state given pacman's direction would show that there are two next states: (0, 1) if pacman goes E and (1, 0) if pacman goes N. From each of these states, pacman goes from (1, 0) to (1, 1) by going E and pacman goes from (0, 1) by going N. If we build a tree from the start state to the end state, it looks like:



Each of the nodes in the tree is a state.

Each of the paths from the start state (shown as the root of the tree) to a leaf is a solution. In this example, there are two solutions that can be shown by listing the actions taken to move between states:

N->E and E->N

The solution can also include the nodes in the tree, where the node can include the state and action information.

Notice that the solution is the action that moves the agent between states. In this example, both solutions are equally good. But, with a more complicated problem, there would be some paths that are shorter than others.

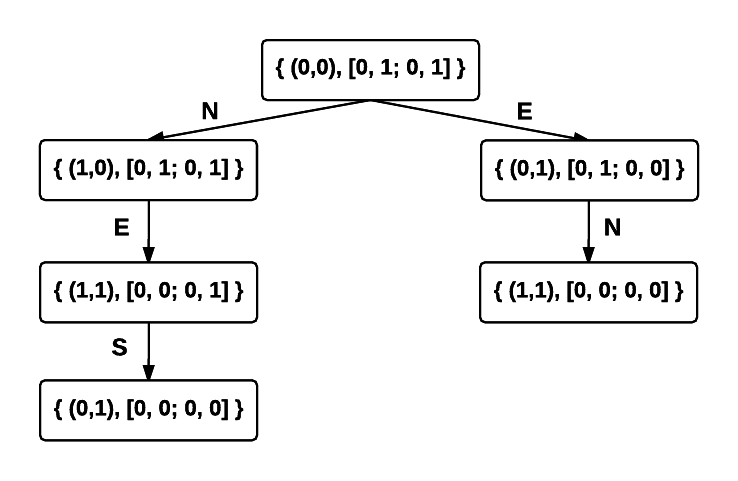
**Example: Eat all of the dots**

**State:** (x,y) location and matrix of dots remaining

**Next state:** current location + step in NSEW direction and matrix of dots remaining after movement

**Goal test:** All dots gone

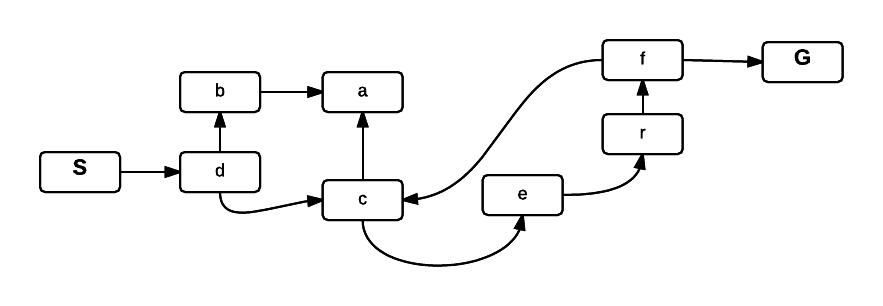
This example also uses the location, but also keeps track of how many dots are remaining. There is still nothing intelligent about how dots are consumed. The successor function moves pacman one step from its current location in a specified direction, and also updates the dot map. Starting from (0,0) in the simple example shown above, the state sequences generated by the successor function would look like:



Neither of these problems include information about the positions of the ghosts. However, if the goal also included staying alive, and the ghosts could attack, then ghost position would also need to be included and the size of the state space would increase.

**State space graph**

Once the state space is defined, the transitions between states can be represented using a state-space graph. This only works on small problems, or sub-sets of larger problems, where the transitions between states can be drawn. In the following directed graph, the vertices are the states and the arrows represent the successor function. Details of the successor function are not specified.



**Question: Find a path from state S to state G.**

There is only one option:

S->d->c->e->r->f->G

**Searching for the path through the graph**

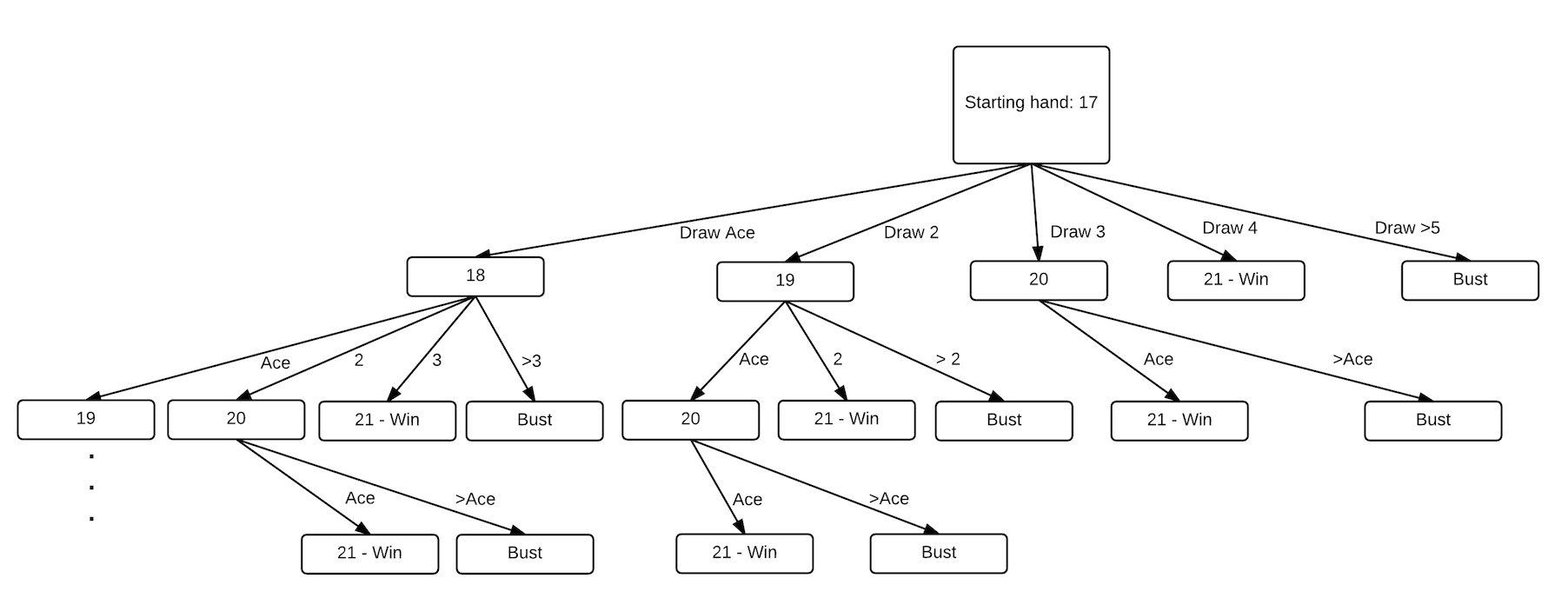
Search algorithms find paths through a graph without having to build the graph first. The states can be represented in a tree, where the current state is a node in the tree, and the next states are children of that node.

Top of the tree, aka the root of the tree, is the start state or current state.

The children of a node are the states that can be reached from that state by one action, or one application of the successor function.

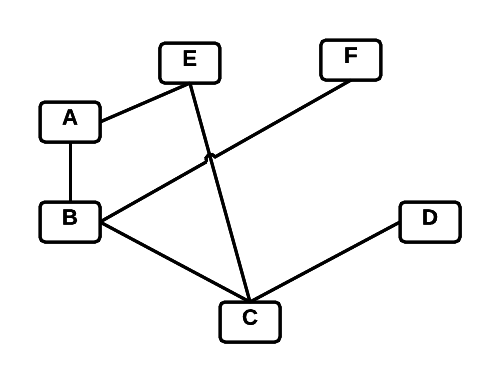
The children's children are two actions from a node, or two applications of a successor function.

The search tree grows exponentially as you move down the tree. Consider, for example, the tree for a move in the game blackjack.

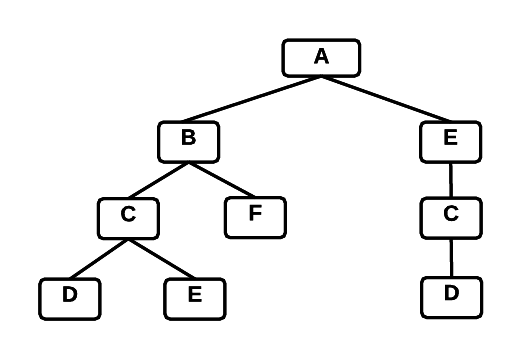


The starting value for the hand is 17, which is given as the root node. The children of the root node show all possible next hands, or states, that can be achieved by a successor function of drawing another card. Two steps from the root are the hands that result from drawing two cards, and so on. Each node in this tree represented a plan for drawing cards and the outcome from that plan. A node that is two steps from the root includes the plan for the first card drawn and the second card drawn.

**Example: In the following graph, starting from A, where can you go in 1, 2, 3 steps? Is there a path from A to D?**



The vertex A becomes the root of the tree, and the vertices connected to A become its children. We can build this tree, starting from vertex A. Nodes are added to the tree if they haven't been visited on the current path.



Where can we go from A in one step? We can determine this by looking at A's children. To know where we can get in two steps, we look at A's children's children. We're only adding nodes to the tree that are not part of the current path.

**Search strategy**

How the tree is traversed, i.e. which states are evaluated and when, to find a solution is set by a search strategy. There are four criteria for evaluating search strategies:

1. Completeness: Is the strategy guaranteed to find a solution if there is one?

2. Time Complexity: How long does it take to find a solution?

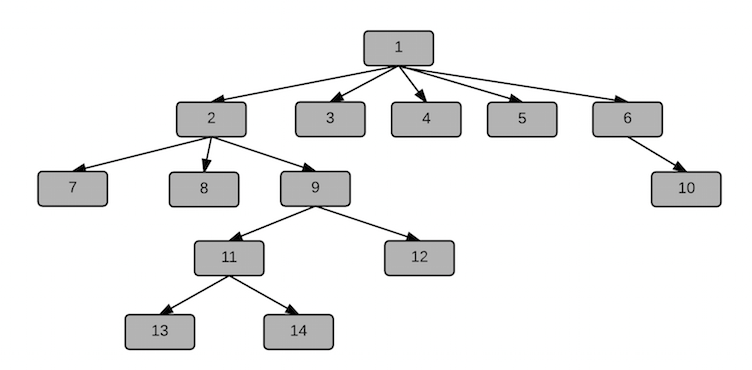
3. Space Complexity: How much memory does the strategy need?

4. Optimality: Does the strategy find the highest quality solution when there are several available?

The first two search strategies we'll discuss are considered uninformed. We have no additional information than the current state and the adjacent states.

**Breadth-first search**

Breadth-first search (BFS) is a search algorithm that identifies solutions by evaluating the vertices in at one level in a graph before going deeper into the graph. If a graph is represented as a tree, the search order of the nodes in the tree is as follows:



In the following BFTraversal algorithm, a queue is used to keep track of the vertices in the graph. When a vertex is visited, its children are enqueued. They can then be dequeued and processed, which involves enqueueing their children. This continues until there are no vertices left to evaluate and the queue is empty.

The BFTraversal algorithm assumes each node has a *visited* property to keep track of whether it has already been evaluated.

BFTraversal(Graph, start)

1.  for each vertex u in Graph.vertices //initialize

2.       u.visited = false

3.  start.visited = true //visited

4. Queue.enqueue(start)

5. while Queue not empty

6. u = Queue.dequeue()

7. for each v in G.Adjacent[u] //each vertex adjacent to u

8. if v.visited == false

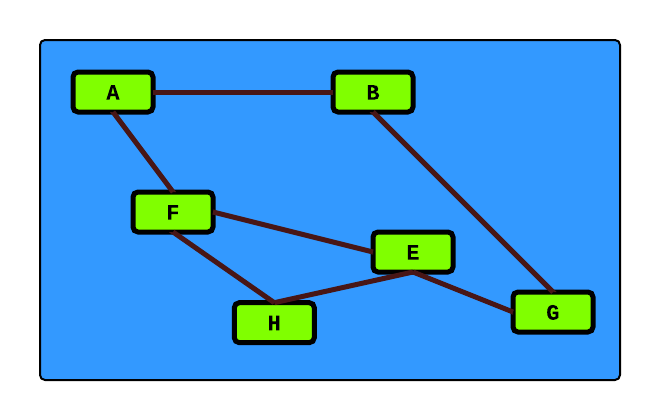
9. v.visited = true

10. output v

11. Queue.enqueue(v) //add v to the queue

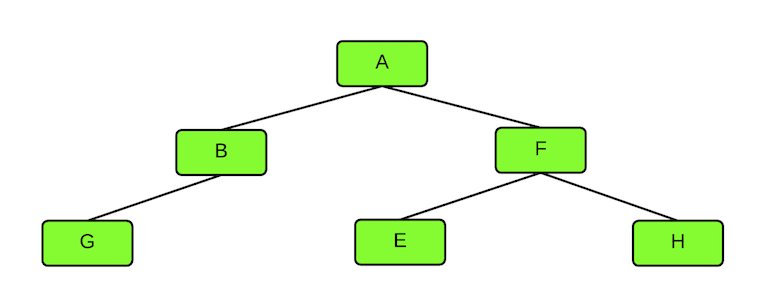
The BFTraversal algorithm traverses the graph and outputs the vertex before it is enqueued.

**Example:** Imagine that the following image shows a bunch of small islands connected by bridges. Starting from A, how many vertices (islands) are visited, and in what order?



Starting from A, we add A to the queue in line 5 of the BFTraversal algorithm. Once it's added, the queue isn't empty, so line 6 is true and we go into the while loop and dequeue A. We loop through each of A's children, which are B and F. If the vertices haven't been visited, we mark each as visited, print its value, and add it to the queue. The next vertex to be dequeued is B, and we then enqueue its unvisited child G. Next, we dequeue the F, and enqueue the E and H. We dequeue the G. There's is nothing left to enqueue for G's children because the E and B have both been marked visited. We dequeue the E, and there's nothing to enqueue. We dequeue the H, and there's nothing to enqueue. The queue is empty and the BFTraversal routine exits.

When viewed as a tree, the search order of BFTraversal looks something like this:



**Shortest path for unweighted graph**

The BFTraversal algorithm above just traverses the graph, we don't get any information about distance from the source to a vertex. For example, how many edges separate A and G on the shortest path? To answer that question, we need to add a distance parameter to each vertex and update it each time a vertex is enqueued. We can modify the pseudo-code above to include distance. In this example, we're also taking an additional input parameter that is the value we're searching for, and returning the vertex once it's found.

BFS(Graph, start, value)

1.  for each vertex in Graph.Vertices

2.       vertex.visited = false

3.  start.visited = true //visited

4. start.distance = 0 //distance to source  
5. start.parent = NULL

6. Queue = empty //initialize empty queue

7. Queue.enqueue(start)

8. while Queue not empty

9. u = Queue.dequeue()

10. for each vertex in Graph.Adjacent[u] //each vertex adjacent to u

11. if vertex.visited == false

12. vertex.distance = u.distance + 1 //u is the parent, dequeued on line 8  
13. vertex.parent = u

14. if vertex.value == value

15. return vertex

16. else

17. vertex.visited = true

18. Queue.enqueue(vertex)

19. return NULL

The BFS algorithm will return the vertex if it's found and NULL if it's not. Stored in the vertex structure is its distance back to the starting vertex *start*.

**BFS Evaluated:**

Completeness: Yes

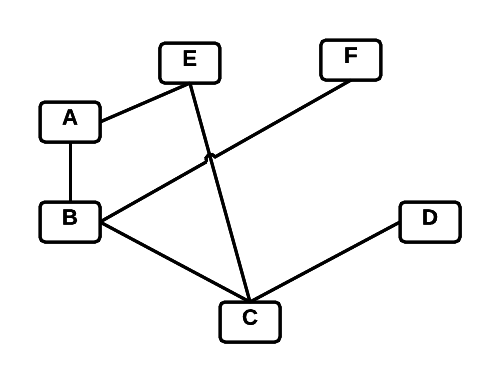
Optimality: If all edge weights the same and the optimal solution is non-decreasing function of depth of node.

Time complexity: O(bd): b is branching factor and d is depth of node.

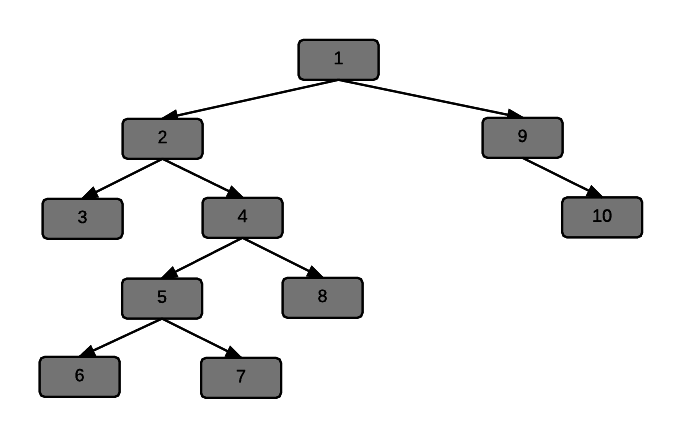
Space complexity: O(bd): b is branching factor and d is depth of node.

**Depth-first search**

Another ordering for searching the vertices in a graph, called depth-first search (DFS), evaluates the vertices along one path before evaluating other paths. DFS is also used in the tree-traversal algorithms for binary trees that print the nodes in the tree. Those algorithms recursively traverse all the way to the leaf nodes in the tree, following the left or right branch, before evaluating any of the other branches in the tree. For the graph shown below, a DFS that starts at vertex A and selects the next adjacent vertex alphabetically, would evaluate vertices in the order *A-B-C-D* backing up and selecting a different path to evaluate vertices *E* and *F*. This ordering differs from a breadth-first search, which would evaluate vertices *B*and *E* before evaluating vertices *C* and *D*.



The evaluation order of vertices in a graph, assuming the graph has been converted to a search tree, is shown here:



Once the bottom of the left branch in a tree is reached, which is equivalent to following a path in a graph until there are no unvisited, adjacent vertices on that path to evaluate, DFS will evaluate all nodes in the right branch. In a graph, following a different branch means selecting a different vertex at the last decision point.

DFS can be handled using a recursive and a non-recursive algorithm. Non-recursive implementations of DFS typically use a stack data structure to store the vertices as they are visited. The stack generates an ordering where the most-recently visited vertices are popped off the stack and processed before vertices that were encountered at higher levels in the tree.

In the recursive version of the algorithm, shown here, the parent of each node is stored when it is visited. There is a potential bug in this algorithm in that we don't know the parent of the original vertex. The algorithm assumes that when DFS is called, vertex.parent has been set elsewhere. The same condition is true for the non-recursive version. In that algorithm, we could set vertex.parent on the first line of the algorithm, but we may or may not want to do that depending on how the algorithm is being used.

DFS(vertex)

    vertex.visited = true

    for each v in vertex.adjacent

        if(!v.visited)

v.parent = vertex  
             print(v.key)

            DFS(v)

depthFirstSearchNonRecursive(vertex)

    vertex.visited = true

    stack.push(vertex)

     while(!stack.isEmpty( ))

          v = stack.pop()

          print(v.key)

          for each x in v.adjacent

              if(!x.visited)

                  x.visited = true

stack.push(x)

**DFS evaluated**

Completeness: yes, if the algorithm includes cycle checking to only visit unvisited nodes.

Optimality: no. It will return a solution if one exists, but not necessarily the optimal one without additional processing.

Time complexity: O(bh): h is height of tree.

Space complexity: O(bh): h is height of the tree. Algorithm only needs to store pointers back to the root.

## Depth-limited search

Depth-limited search is a DFS algorithm that only searches to a fixed depth. Limiting how deep the search goes solves the problem on going down a canyon when the solution is in a node much higher up in the tree.

Depth-limited search specifies a depth, L, and when L is reached, it is treated as the bottom of the tree, and successor nodes of L are not evaluated. Having a depth limit solves the infinite depth problem, when cycle checking not included. However, new problems are introduced.

When L < D, where D is the depth of the solution, then depth-limited search will not find the solution. Therefore, when L < D, the algorithm is not complete.

When L > D, there could be a solution found deeper in the tree than a better solution at depth D on another branch of the tree.

Complete: Yes, but only when L >= D. Not when L < D

Optimal: No.

Time complexity: O(bL)

Space complexity: O(bL)

DLS Algorithm

The parameters for DLS are the current vertex being evaluated, the goal, and the current depth of the search. Each time DLS is called, depth is decreased by 1 and if depth reaches 0 before a solution is found, then the algorithm returns cutoff, which is a constant set elsewhere. After DLS returns a result, if the result is the search value, then the vertex is returned. Otherwise, additional recursive calls are made.

DLS(vertex, goal, depth)

    if vertex.key == goal  
 return vertex  
 if depth == 0  
 return cutoff //some value for cutoff needs to be set  
 vertex.visited = true  
 cutoff\_occurred = false

    for each v in vertex.adjacent

        if(!v.visited)

v.parent = vertex

            result = DLS(v, goal, depth--)  
 if result == cutoff  
 cutoff\_occurred = true  
 else if result != failure  
 return result  
 if cutoff\_occurred  
 return cutoff  
 else  
 return failure

## Iterative deepening DFS

In iterative deepening DFS, the maximum depth of the search is increased by 1 and the search is repeated. The downside to Iterative Deepening DFS is that the search restarts each time from the root of the tree, which results in repeat computation.

Complete: yes

Optimal: yes

Time complexity: O(bd), where d is solution depth.

Space complexity: O(bd)

IterativeDeepening(vertex, goal, maxDepth)

     for x = 0 to maxDepth

          result = DLS(vertex, goal, x)

          if result is not failure or cutoff

               return result

     return failure

## Uniform Cost Search

BFS finds the optimal solution when all step costs are equal by expanding the shallowest node. In a weighted graph, the optimal solution is the one with the lowest path weight to the goal node. The Uniform Cost Search (UCS) algorithm is used to find the shortest path in a weighted graph and works by expanding the node with the lowest path cost overall. The UCS algorithm doesn't count the number of steps, but rather the path cost of the steps. Provided that the edge weights are larger than a specified value, then the algorithm will find the optimal solution.

Complete: yes, if all edges have positive weight > ε.

Optimal: yes

Time complexity:

Every edge has a minimum weight ε.

C is optimal solution cost.

Max path length cannot exceed C/ε.

O(bC/ε)

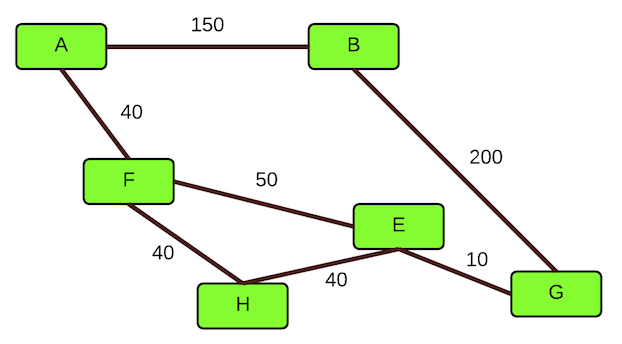
Space complexity:

O(bC/ε)

One implementation of UCS is known Dijkstras algorithm.

### Dijkstra's algorithm

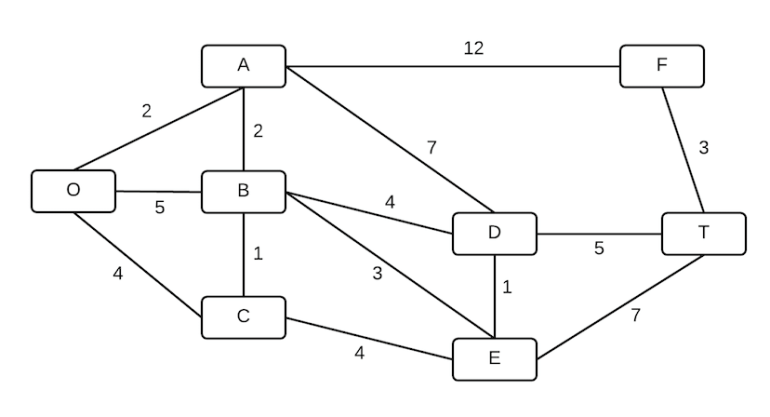
Consider this graph:



There are multiple paths between A and G. One path goes A->B->G and has a weight of 350. Another path goes A->F->E->G and has a weight of 100. Finally, there is a path that goes A->F->H->E->G with a weight of 130. The shortest path in this weighted graph is the one with a weight of 100: A->F->E->G.

The path with the shortest distance in a weighted graph, where all weights are greater than 0, can be found with Dijkstra's algorithm.

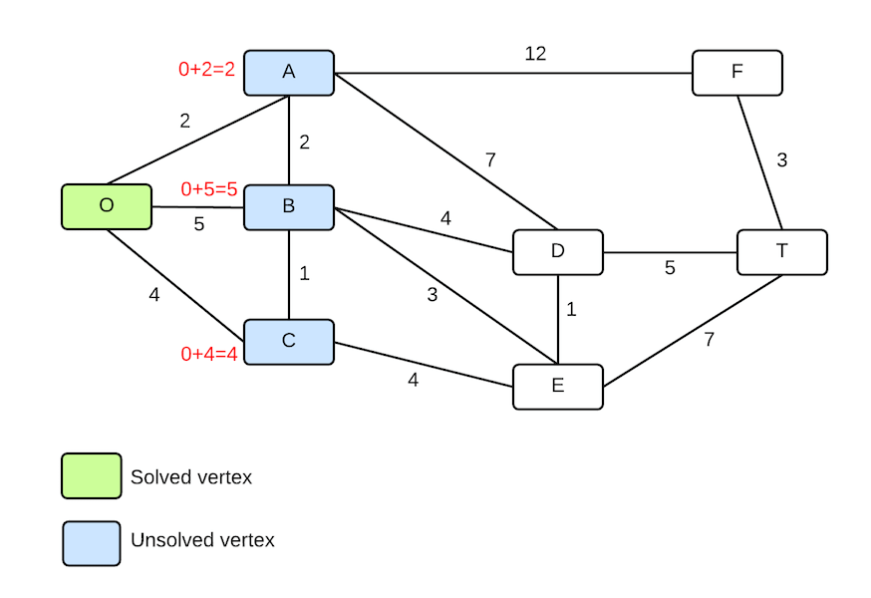
**Dijkstra's Example: Find the shortest distance through this graph between the O vertex and the T vertex:**



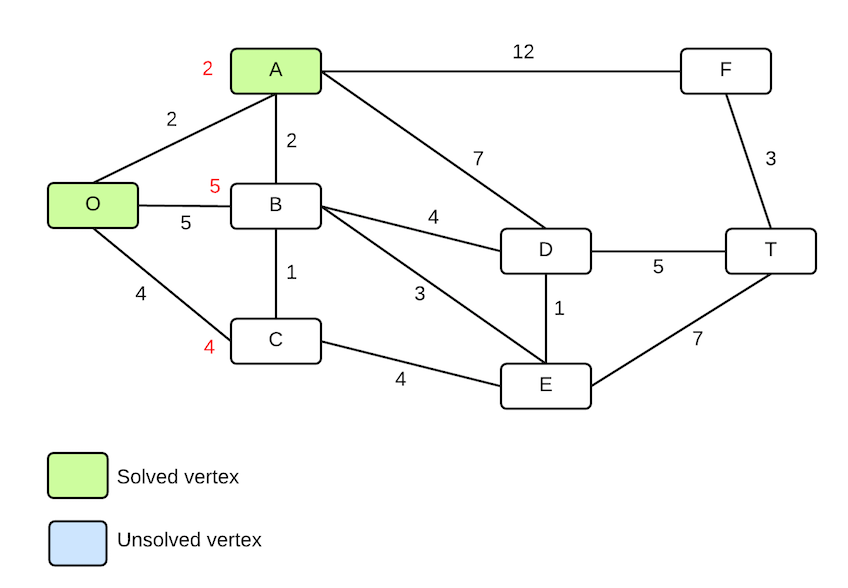
Dijkstra's algorithm is a greedy algorithm, it chooses the best path on each iteration and builds the best overall solution from these individual best decisions. The best path on any iteration is the shortest path to any vertex that hasn't yet been visited.

To start, mark the origin as solved and the distance to the origin as 0.

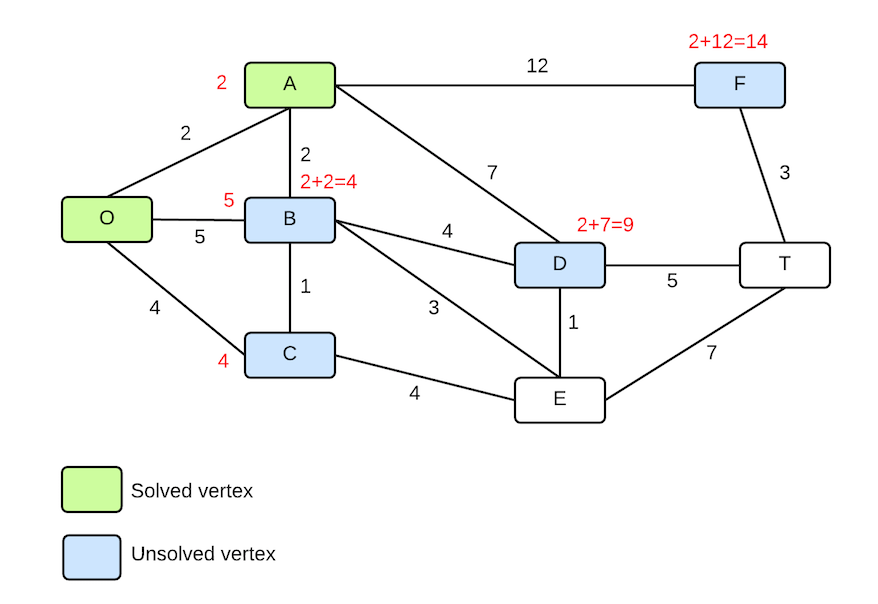
Next, find the unsolved vertices adjacent to the origin and calculate the distance to those vertices using the distance to the origin + the edge weight connecting the vertex to the origin.



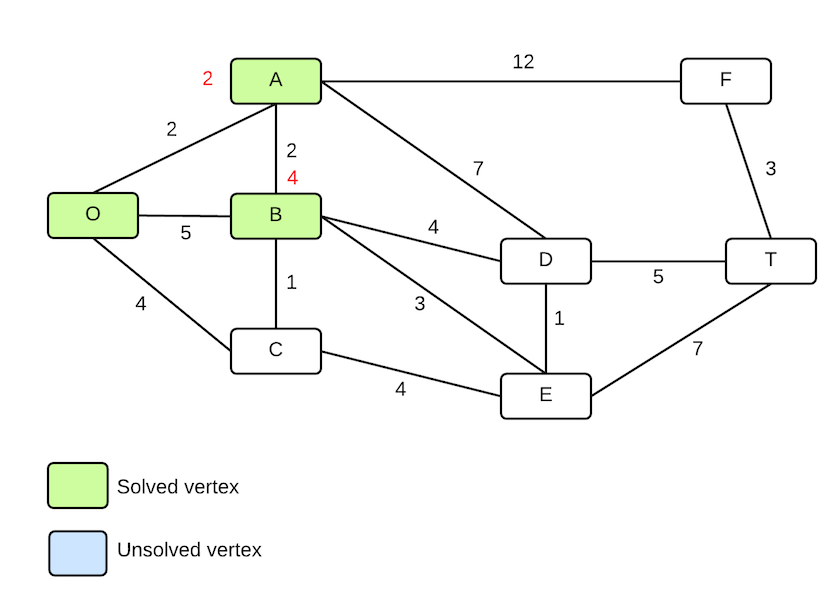
Select the vertex with the shortest distance and mark that vertex as solved. Update the vertex to show the distance to the vertex from the graph origin. In this example, the shortest distance is to the A vertex. The A is now solved, with a distance of 2.



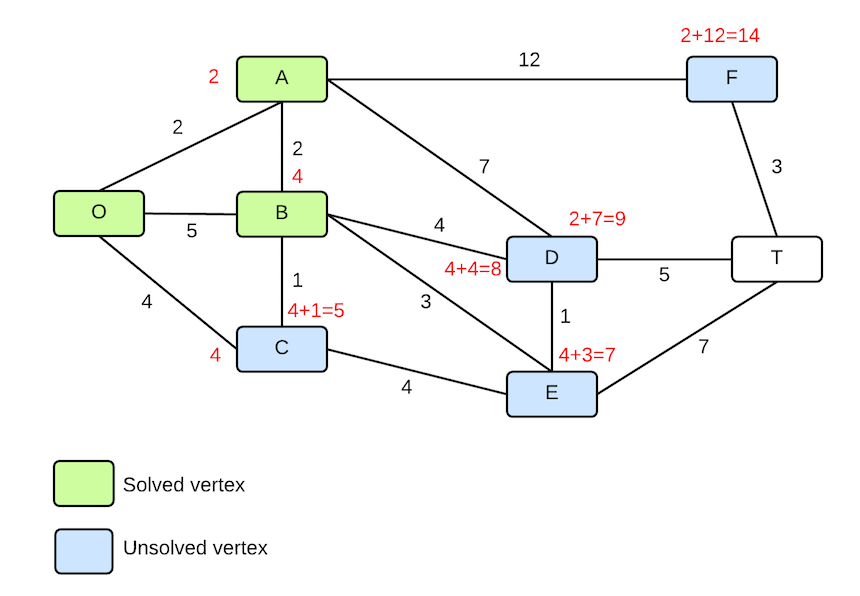
Next, repeat the process of selecting all unsolved vertices adjacent to solved vertices, and calculate the distance to each vertex. The unsolved vertices from the O and A are B, C, D, and F.



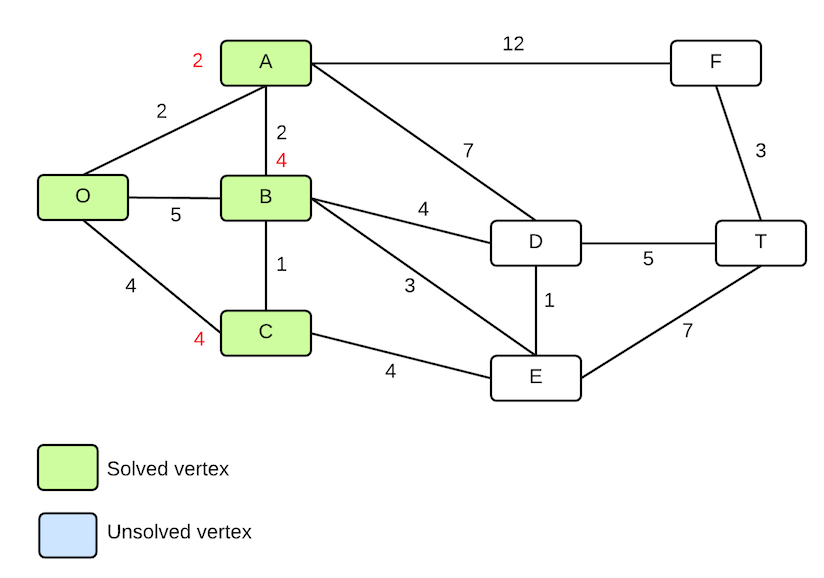
There is a tie for the shortest. The distance to the B from A is 2 + 2 = 4. The distance from the O to C is 0 + 4 = 4. Use a tie-breaker algorithm to select one of them. I'll select the B, mark it as solved, and update its distance to 4. We now know that the shortest distance from the origin to B is 4.



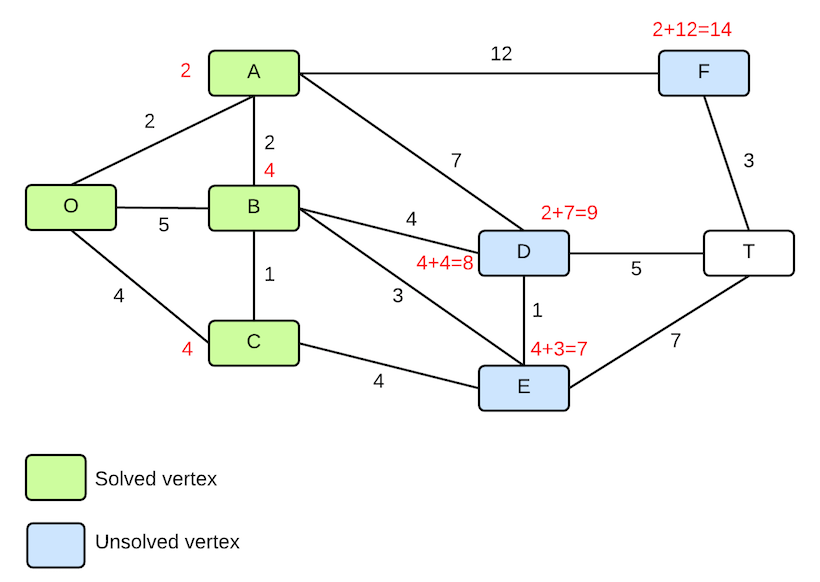
Next, select all unsolved vertices adjacent to all solved vertices and calculate their distances. The unsolved vertices are C, with a distance of 4, E with a distance of 7, D with a distance of 8 from the B and 9 from the A, and F, with a distance of 14.



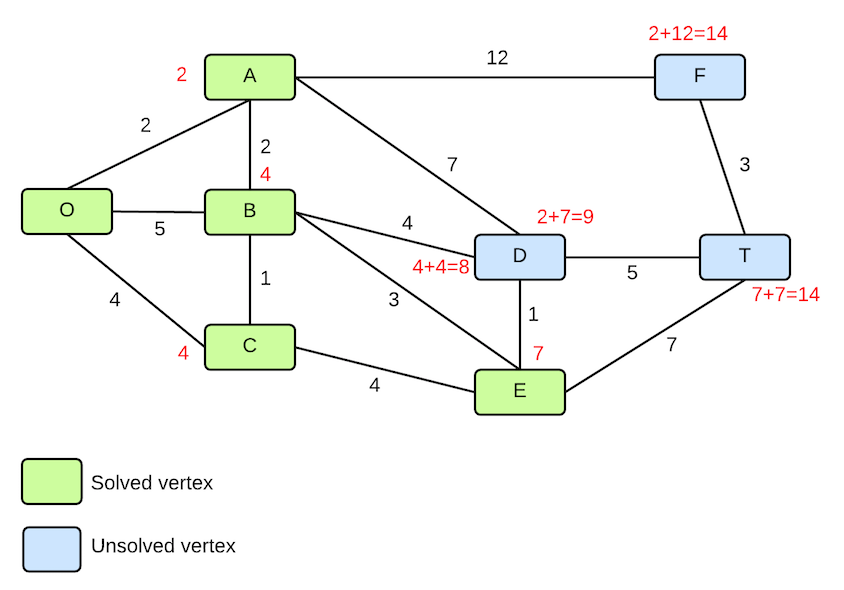
The shortest distance is the C. Mark it as solved and update its distance.

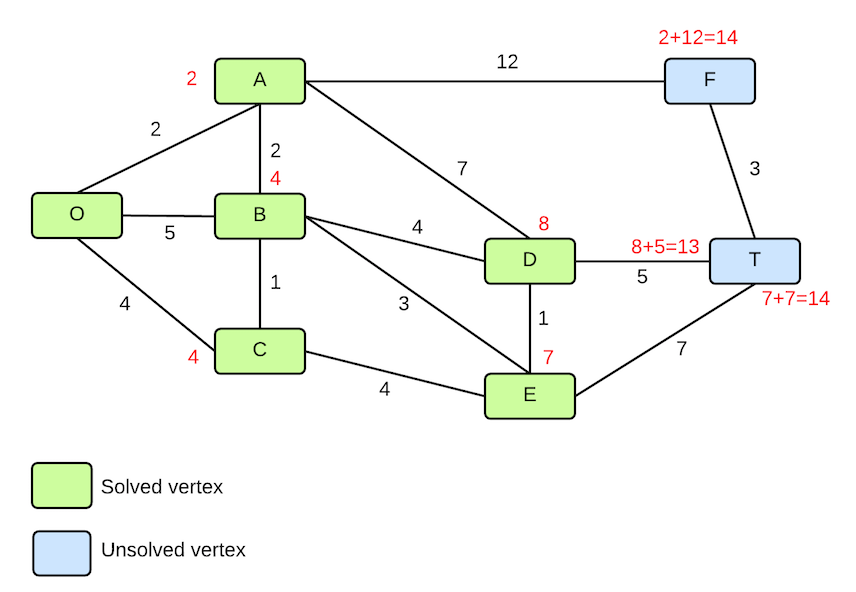


Repeat, selecting all unsolved vertices from all solved vertices. Notice that all vertices connected to the origin have been marked as solved, so there is nothing to calculate from the origin for this iteration. The adjacent vertices are D, E, and F. The shortest distance is to E with a distance of 7.



Mark the E as solved. Calculate distance to all unsolved vertices adjacent to solved vertices. The shortest distance is to the D, either coming from the B with a distance of 4 + 4 = 8, or from the E with a distance of 7 + 1 = 8. Select one of the paths. I'll select the B, and mark the D as solved.



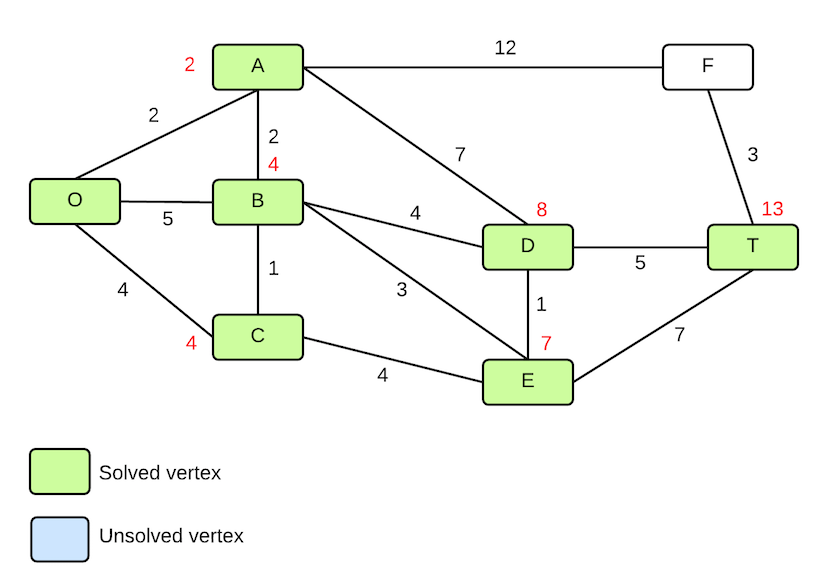
Calculate distances to unsolved vertices. Here, we see that the destination vertex T now has a connection from both the D with a distance of 13 and the E with a distance of 14. 

Select the path from the D. Mark the T as solved, and the final distance from the origin O to the distance T is 13. There are multiple paths that produce the distance:

O->A->B->D->T

O->A->B->E->D->T

O->C->E->D->T



Dijkstra(start, end)

    startV = search(start)

    endV = search(end)

    startV.solved = true

    startV.distance = 0

    solved = {startV} //list of solved vertices

    while (!endV.solved)

        minDistance = INT\_MAX

        solvedV = NULL

        for s in solved

          for y in s.adjacent

              if(!y.solved)

                  dist = s.distance + y.distance

                  if(dist < minDistance)

                      solvedV = y

                      minDistance = dist

                      parent = s

    solvedV.distance = minDistance

    solvedV.parent = parent

    solvedV.solved = true

    solved.add(solvedV)

# Informed Search

Unlike BFS and DFS which consider only the current state information to expand nodes in the search trees, informed search methods use problem-specific knowledge to find the solution to a problem more efficiently. The informed search algorithms we're going to look at use an approach called best-first search, where a node is selected for evaluation based on an evaluation function, f(n). The evaluation function is considered a cost estimate, and it's the node with the lowest cost that we want to evaluate. We saw this approach with UCS. The difference between UCS and what we'll see with informed search algorithms is how f(n) is calculated.

The evaluation function f(n), includes a heuristic function h(n),that is an estimate of the cost to the goal from the current node.

**Heuristic function definition:**

h(n) = estimated cost of cheapest path from state at node n to goal state.

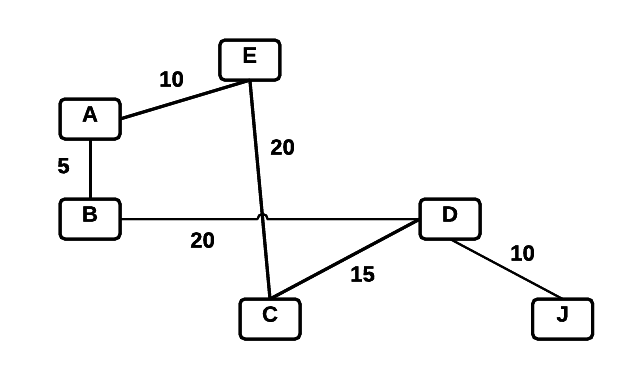
None of the uninformed search algorithms we looked at used any estimation about how to find the solution. The heuristic is an educated guess given knowledge of the domain. There are good heuristics and bad heuristics and finding a good heuristic is very important if you want to find a good solution.

## Greedy best-first search

Greedy best-first search is an informed search algorithm that evaluates the node that is the closest to the goal using just the heuristic function for evaluation. This algorithm can actually produce very bad solutions because it can easily get stuck choosing a path that appears to be the best, but is ultimately longer, infinite, or not complete. The problem with greedy best-first search is that it only evaluates nodes from the parent, not all the way back to the root of the search. Therefore, once a sub-optimal decision is made, it can't necessarily be undone. This is the greedy part of the algorithm, it only considers the locally best solution.

**Example: Find the shortest path from A to C.**

Heuristic: straight line distance between two places. Using this heuristic means that the best move is the one that gets you closest to the goal.



The straight line distances between the vertices in the graph and the C vertex are (approximately, these numbers are made up but appear to be about right from looking at the graph.):

A-C = 17

B-C = 15

C-C = 0

D-C = 15

E-C = 20

J-C = 20

Starting at A, the nodes adjacent to A are evaluated using the straight line distance to the goal, and the objective is to select the minimum value:

f(B) = 15

f(E) = 20

B is closer to the goal, so move to B. From B, select the node closest to the goal. There is only one option from B, which is D, and f(D) = 20.

From D, go to C with a cost of 15.

The path that greedy Best-First search finds is B-D-C with a cost of 40. However, there is a shorter path that goes A-E-C with a cost of 30. Because the path appears to be longer at the first step, greedy search won't select it.

**Another example, where greedy best-first search produces optimal solution**

**Find the shortest path from A to J**

The straight line distance for this example are:

A-J = 35

B-J = 30

C-J = 20

D-J = 10

E-J = 35

J-J = 0

The nodes evaluated from A are B and E, where f(B) = 30 and f(E) = 35. Node B is selected. From B, there is only one choice D, and from D, the options are J and C. f(J) =  0 and f(C) = 20. The final path is A-B-D-J, with a cost of 35, which is also the shortest path.

## A\* Search

A\* is a classic best-first search algorithm that combines a heuristic algorithm with observed values. The algorithm uses both the cost to reach a node, g(n), and the estimated cost to get from the node to the goal, h(n). The value of g(n) is the observed cost and the value of h(n) is the estimated cost using the heuristic. The value of a node, f(n), is calculated as:

f(n) = g(n) + h(n)

This algorithm is similar to UCS, except that it uses f(n) = g(n) + h(n) instead of f(n) = g(n).

### ****A\* Algorithm****

**AStar(Start, End)**

Open.append(Start)

Closed = empty list

while Open not empty

   node = minimum-cost node in Open

   Open.remove(node)

   if node != End

       Closed.append(node)

       for n in node.adjacent

           if isValid(n) //not closed and not a wall or other invalid state

               calculate f(n)

               n.parent = node

               if n in Open

                   Replace if f(n) lower than existing f(n) for n

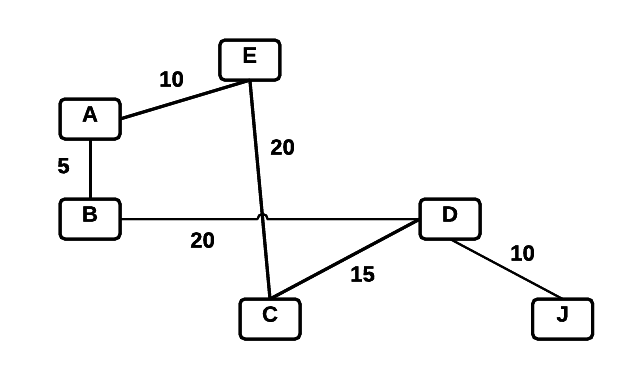
               else

                   Open.append(n)

   else

       break

**Example: find the shortest path from A to C**



Use the same straight line distances as we used previously.

A-C = 17

B-C = 15

C-C = 0

D-C = 15

E-C = 20

J-C = 20

Evaluate nodes adjacent to A:

f(B) = distance to B + heuristic for B = 5 + 15 = 20

f(E) = 10 + 20 = 30 (for E, the straight line distance heuristic is also the edge weight)

B is still a better choice than E, but we don't forget about E.

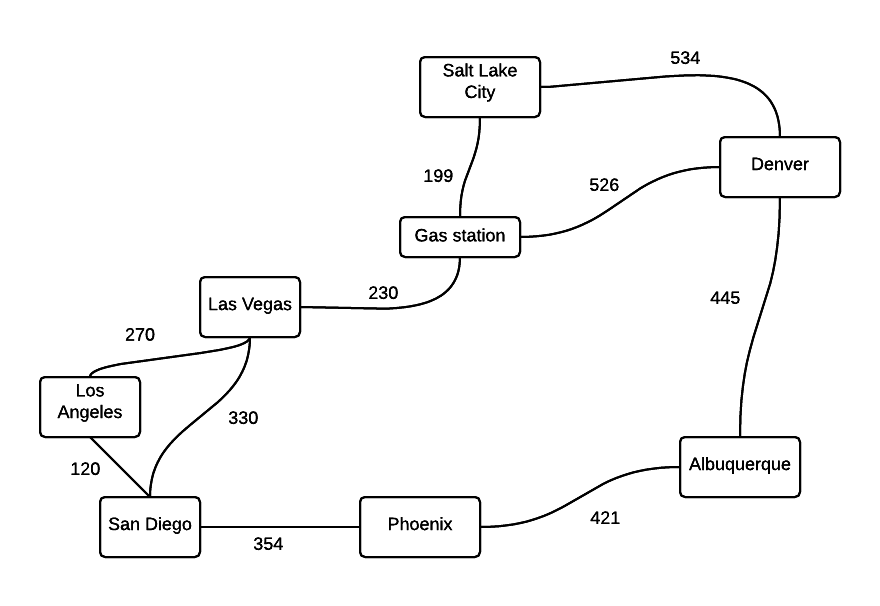
From B, we evaluate D since that's the only link to B.

f(D) = distance to D + straight line distance = 25 + 15 = 40.

The score for 40 is greater than f(E), so we pick up the search from E. The next step from E is to C, with a distance of 30.

The path that A\* produces is the optimal one: A-E-C with a distance of 30.

**Example: find shortest distance from San Diego to Denver in the following map.**



h(n) = straight line distance to goal city is the heuristic.

We first need to know the straight line distances from each city in the map to Denver.

LA - Denver = 831

Las Vegas - Denver = 605

Phoenix - Denver = 586

Gas station - Denver = 423

Salt Lake City - Denver = 371

Albuquerque - Denver = 349

Start in San Diego and evaluate the adjacent cities to SD to determine where to go first. There are three options, Los Angeles, Las Vegas, and Phoenix.

f(LA) = distance SD to LA + straight line distance LA to Denver = 120 + 831 = 1051.

f(Vegas) = 330 + 605 = 935

f(Phoenix) = 354 + 586 = 940

The path with the shortest estimated cost is the one that first goes to Las Vegas. The algorithm then evaluates the shortest path starting from Las Vegas. There is only one option from Vegas, go to the Gas station, which has a cost of

f(Gas station) = cost to gas station + heuristic cost to Denver.

The cost to the gas station is 330 + 230, which is the cost to Vegas + the cost from Vegas to the gas station.

f(Gas station) = 560 + 423 = 983

The paths being evaluated are:

f(Gas station) = 983

f(Phoenix) = 940

f(LA) = 1051

The path through Phoenix now has the shortest cost, so that path is expanded next. The only next step from Phoenix is to Albq., which has a score of:

f(Albq.) = (354+421) + 349 = 1124

The paths being evaluated now are:

f(Gas station) = 983

f(LA) = 1051

f(Albq.) = 1124

The gas station is the shortest path. There are two next steps from the gas station to evaluate:

f(Salt Lake) = 759+371 = 1130

f(Denver) = 560 + 526 + 0 = 1086

The path to Denver is shorter than the path going through Salt Lake City, so we have found the shortest path. The path goes: San Diego -> Vegas -> Gas station -> Denver.

Evaluating A\*

Complete: yes, as long as heuristic satisfies certain conditions

Optimal: yes, as long as heuristic satisfies certain conditions

Time complexity: O(bd)

Space complexity: O(bd)

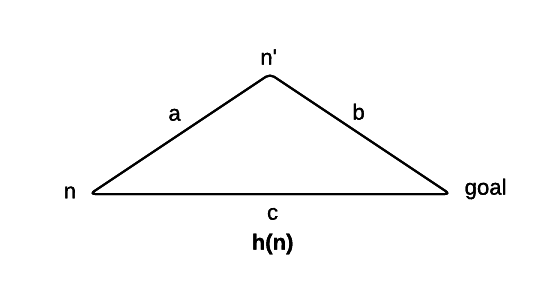
## 

## Admissible Heuristics

A\* is optimal if h(n) is admissible and consistent.

An admissible heuristic never overestimates the cost to reach the goal. We used the straight-line distance is the example as the straight line distance is always the shortest distance between two points.

A heuristic is consistent if for every node n and every successor n' of n, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'. This is an application of the triangle inequality.



h(n) = c

a + b >= c for admissibility and consistency.

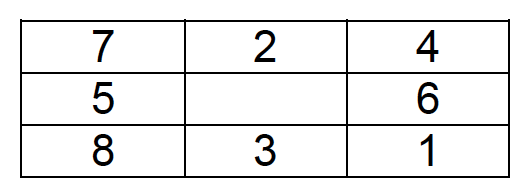
a is the step cost n to n'.

b is the cost of n' to the goal.

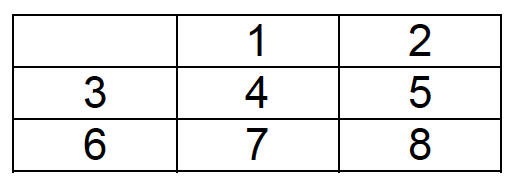
h(n) <= a + h(n')

## Good heuristics

Finding a good solution using A\* search requires a good heuristic. Consider the 8 puzzle, where the objective is to re-arrange the tiles from a mixed-up order into a sorted order.



Start state



goal state

**Possible heuristics:**

1. The number of tiles out of place. In this start state, all 8 tiles are out of place.

h1 = number of tiles out of place

h1 = 8

2. Sum of the distances of tiles from the goal position. Each move is vertical or horizontal, no diagonal moves. In this heuristic, we assume that tiles are not blocked by other tiles. The Manhattan distance = horizontal + vertical moves.

tile 1 = 2v + 1h = 3

tile 2 = 1h = 1

tile 3 = 1h + 1v = 2

tile 4 = 1h + 1v = 2

tile 5 = 2h = 2

tile 6 = 2h + 1v = 3

tile 7 = 2v + 1h = 3

tile 8 = 2h = 2

The sum of these distances is 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18

Is h2 always greater than or equal to h1? Why or why not?

Yes. Any tile out of place will require at least one move to put it in the correct position. If h1 = 1, then h2 >= 1. If h1 = 2, then h2 >= 2.

For any n, it is true that h2(n) >= h1(n). We say that h2 dominates h1. A heuristic with a larger value that doesn't overestimate the solution cost is better than a heuristic with lower values. We want a heuristic that is as close to the actual solution as possible.

### Inventing admissible heuristics

In the 8-tile problem, h1 and h2 are estimates of the remaining path length to solve the puzzle. They are also the exact path length for a simplified version of the problem where existing constraints on the problem are removed.

h1 is the path length if tiles could move more than one square. There would be h1 swaps of tiles in the puzzle.

h2 is the path length if tiles could move into occupied squares.

The existing problem with constraints removed is known as a **relaxed problem**. The optimal solution to the relaxed problem is a good heuristic for the original problem.

**Formal definition of the 8-tile problem:**

A tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank. Removing constraints generates three relaxed problems:

1. A tile can move from square A to B if A is adjacent to B. This is the Manhattan distance heuristic, the blank constraint is removed.

2. A tile can move from square A to square B is B is blank. This removes the adjacency constraint.

3. A tile can move from square A to square B. Both constraints are removed, and the number of misplaced tiles heuristic is used.

# A\* Pathfinding for Beginners

By Patrick Lester (Updated July 18, 2005)

*This article has been translated into*[*Albanian*](http://www.policyalmanac.org/games/aStarTutorial_al.pdf)*,*[*Chinese*](http://www.policyalmanac.org/games/Chine%20Translation%20-%20For%20beginners.html)*,*[*Finnish*](http://www.policyalmanac.org/Finnish/astar_fin.pdf)*,*[*German*](http://www.policyalmanac.org/games/aStarTutorial_de.html)*,*[*Greek*](http://www.policyalmanac.org/games/aStarTutorial_greek.htm)*,*[*Korean*](http://cozycoz.egloos.com/9748811)*,*[*Polish*](http://www.policyalmanac.org/games/aStarTutorial_pol.htm)*,*[*Portuguese*](http://www.policyalmanac.org/games/aStarTutorial_port.htm)*,*[*Romanian*](http://webhostinggeeks.com/science/pathfinding-for-rm)*,*[*Russian*](http://www.policyalmanac.org/games/aStarTutorial_rus.htm)*,*[*Serbian*](http://www.policyalmanac.org/games/aStarTutorial_rs.html)*, and*[*Spanish*](http://www.policyalmanac.org/games/articulo1.htm)*. Other translations are welcome. See email address at the bottom of this article.*

The A\* (pronounced A-star) algorithm can be complicated for beginners. While there are many articles on the web that explain A\*, most are written for people who understand the basics already. This article is for the true beginner.

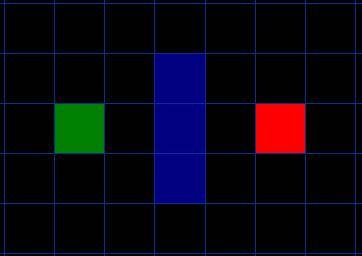
This article does not try to be the definitive work on the subject. Instead it describes the fundamentals and prepares you to go out and read all of those other materials and understand what they are talking about. Links to some of the best are provided at the end of this article, under Further Reading.

Finally, this article is not program-specific. You should be able to adapt what's here to any computer language. As you might expect, however, I have included a link to a sample program at the end of this article. The sample package contains two versions: one in C++ and one in Blitz Basic. It also contains executables if you just want to see A\* in action.

But we are getting ahead of ourselves. Let's start at the beginning ...

**Introduction: The Search Area**

Let’s assume that we have someone who wants to get from point A to point B. Let’s assume that a wall separates the two points. This is illustrated below, with green being the starting point A, and red being the ending point B, and the blue filled squares being the wall in between.

    
[Figure 1]

The first thing you should notice is that we have divided our search area into a square grid. Simplifying the search area, as we have done here, is the first step in pathfinding. This particular method reduces our search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, and its status is recorded as walkable or unwalkable. The path is found by figuring out which squares we should take to get from A to B. Once the path is found, our person moves from the center of one square to the center of the next until the target is reached.

These center points are called “nodes”. When you read about pathfinding elsewhere, you will often see people discussing nodes. Why not just call them squares? Because it is possible to divide up your pathfinding area into something other than squares. They could be rectangles, hexagons, triangles, or any shape, really. And the nodes could be placed anywhere within the shapes – in the center or along the edges, or anywhere else. We are using this system, however, because it is the simplest.

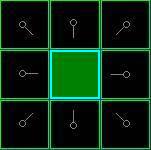
**Starting the Search**

Once we have simplified our search area into a manageable number of nodes, as we have done with the grid layout above, the next step is to conduct a search to find the shortest path. We do this by starting at point A, checking the adjacent squares, and generally searching outward until we find our target.

We begin the search by doing the following:

1. Begin at the starting point A and add it to an “open list” of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later. It contains squares that might fall along the path you want to take, but maybe not. Basically, this is a list of squares that need to be checked out.
2. Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too. For each of these squares, save point A as its “parent square”. This parent square stuff is important when we want to trace our path. It will be explained more later.
3. Drop the starting square A from your open list, and add it to a “closed list” of squares that you don’t need to look at again for now.

At this point, you should have something like the following illustration. In this illustration, the dark green square in the center is your starting square. It is outlined in light blue to indicate that the square has been added to the closed list. All of the adjacent squares are now on the open list of squares to be checked, and they are outlined in light green. Each has a gray pointer that points back to its parent, which is the starting square.

   
  [Figure 2]

Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

**Path Scoring**

The key to determining which squares to use when figuring out the path is the following equation:

F = G + H

where

* G = the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.
* H = the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, which can be a bit confusing. The reason why it is called that is because it is a guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). You are given one way to calculate H in this tutorial, but there are many others that you can find in other articles on the web.

Our path is generated by repeatedly going through our open list and choosing the square with the lowest F score. This process will be described in more detail a bit further in the article. First let’s look more closely at how we calculate the equation.

As described above, G is the movement cost to move from the starting point to the given square using the path generated to get there. In this example, we will assign a cost of 10 to each horizontal or vertical square moved, and a cost of 14 for a diagonal move. We use these numbers because the actual distance to move diagonally is the square root of 2 (don’t be scared), or roughly 1.414 times the cost of moving horizontally or vertically. We use 10 and 14 for simplicity’s sake. The ratio is about right, and we avoid having to calculate square roots and we avoid decimals. This isn’t just because we are dumb and don’t like math. Using whole numbers like these is a lot faster for the computer, too. As you will soon find out, pathfinding can be very slow if you don’t use short cuts like these.

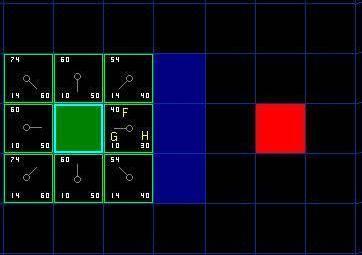
Since we are calculating the G cost along a specific path to a given square, the way to figure out the G cost of that square is to take the G cost of its parent, and then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that parent square. The need for this method will become apparent a little further on in this example, as we get more than one square away from the starting square.

H can be estimated in a variety of ways. The method we use here is called the Manhattan method, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically. This is (probably) called the Manhattan method because it is like calculating the number of city blocks from one place to another, where you can’t cut across the block diagonally.

Reading this description, you might guess that the heuristic is merely a rough estimate of the remaining distance between the current square and the target "as the crow flies." This isn't the case. We are actually trying to estimate the remaining distance along the path (which is usually farther). The closer our estimate is to the actual remaining distance, the faster the algorithm will be. If we overestimate this distance, however, it is not guaranteed to give us the shortest path. In such cases, we have what is called an "inadmissible heuristic."

Technically, in this example, the Manhattan method is inadmissible because it slightly overestimates the remaining distance. But we will use it anyway because it is a lot easier to understand for our purposes, and because it is only a slight overestimation. On the rare occasion when the resulting path is not the shortest possible, it will be nearly as short. Want to know more? You can find equations and additional notes on heuristics [here](http://www.policyalmanac.org/games/heuristics.htm).

F is calculated by adding G and H. The results of the first step in our search can be seen in the illustration below. The F, G, and H scores are written in each square. As is indicated in the square to the immediate right of the starting square, F is printed in the top left, G is printed in the bottom left, and H is printed in the bottom right.

   
  [Figure 3]

So let’s look at some of these squares. In the square with the letters in it, G = 10. This is because it is just one square from the starting square in a horizontal direction. The squares immediately above, below, and to the left of the starting square all have the same G score of 10. The diagonal squares have G scores of 14.

The H scores are calculated by estimating the Manhattan distance to the red target square, moving only horizontally and vertically and ignoring the wall that is in the way. Using this method, the square to the immediate right of the start is 3 squares from the red square, for a H score of 30. The square just above this square is 4 squares away (remember, only move horizontally and vertically) for an H score of 40. You can probably see how the H scores are calculated for the other squares.

The F score for each square, again, is simply calculated by adding G and H together.

# Continuing the Search

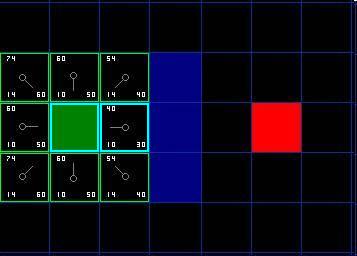
To continue the search, we simply choose the lowest F score square from all those that are on the open list. We then do the following with the selected square:

4) Drop it from the open list and add it to the closed list.

5) Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain), add squares to the open list if they are not on the open list already. Make the selected square the “parent” of the new squares.

6) If an adjacent square is already on the open list, check to see if this path to that square is a better one. In other words, check to see if the G score for that square is lower if we use the current square to get there. If not, don’t do anything.   
    On the other hand, if the G cost of the new path is lower, change the parent of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected square). Finally, recalculate both the F and G scores of that square. If this seems confusing, you will see it illustrated below.

Okay, so let’s see how this works. Of our initial 9 squares, we have 8 left on the open list after the starting square was switched to the closed list.  Of these, the one with the lowest F cost is the one to the immediate right of the starting square, with an F score of 40. So we select this square as our next square. It is highlight in blue in the following illustration.

    
[Figure 4]

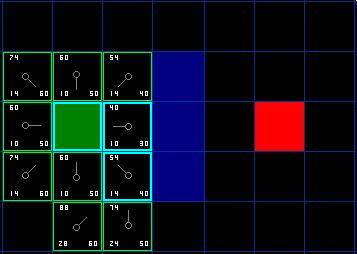
First, we drop it from our open list and add it to our closed list (that’s why it’s now highlighted in blue).  Then we check the adjacent squares. Well, the ones to the immediate right of this square are wall squares, so we ignore those. The one to the immediate left is the starting square. That’s on the closed list, so we ignore that, too.

The other four squares are already on the open list, so we need to check if the paths to those squares are any better using this square to get there, using G scores as our point of reference. Let’s look at the square right above our selected square. Its current G score is 14. If we instead went through the current square to get there, the G score would be equal to 20 (10, which is the G score to get to the current square, plus 10 more to go vertically to the one just above it). A G score of 20 is higher than 14, so this is not a better path. That should make sense if you look at the diagram. It’s more direct to get to that square from the starting square by simply moving one square diagonally to get there, rather than moving horizontally one square, and then vertically one square.

When we repeat this process for all 4 of the adjacent squares already on the open list, we find that none of the paths are improved by going through the current square, so we don’t change anything. So now that we looked at all of the adjacent squares, we are done with this square, and ready to move to the next square.

So we go through the list of squares on our open list, which is now down to 7 squares, and we pick the one with the lowest F cost. Interestingly, in this case, there are two squares with a score of 54. So which do we choose? It doesn’t really matter. For the purposes of speed, it can be faster to choose the last one you added to the open list. This biases the search in favor of squares that get found later on in the search, when you have gotten closer to the target. But it doesn’t really matter. (Differing treatment of ties is why two versions of A\* may find different paths of equal length.)

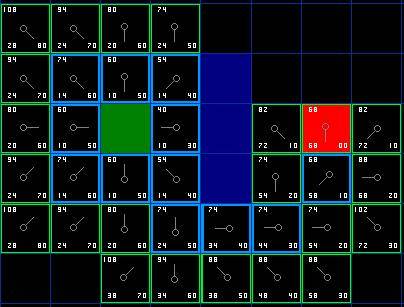
So let’s choose the one just below, and to the right of the starting square, as is shown in the following illustration.

    
[Figure 5]

This time, when we check the adjacent squares we find that the one to the immediate right is a wall square, so we ignore that. The same goes for the one just above that. We also ignore the square just below the wall. Why? Because you can’t get to that square directly from the current square without cutting across the corner of the nearby wall. You really need to go down first and then move over to that square, moving around the corner in the process. (Note: This rule on cutting corners is optional. Its use depends on how your nodes are placed.)

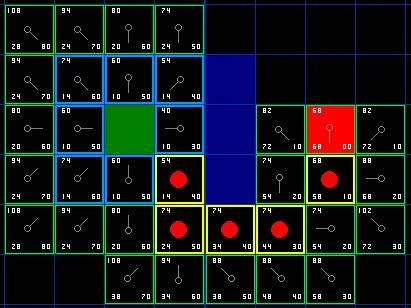
That leaves five other squares.  The other two squares below the current square aren’t already on the open list, so we add them and the current square becomes their parent. Of the other three squares, two are already on the closed list (the starting square, and the one just above the current square, both highlighted in blue in the diagram), so we ignore them. And the last square, to the immediate left of the current square, is checked to see if the G score is any lower if you go through the current square to get there. No dice. So we’re done and ready to check the next square on our open list.

We repeat this process until we add the target square to the closed list, at which point it looks something like the illustration below.

    
[Figure 6]

Note that the parent square for the square two squares below the starting square has changed from the previous illustration. Before it had a G score of 28 and pointed back to the square above it and to the right. Now it has a score of 20 and points to the square just above it. This happened somewhere along the way on our search, where the G score was checked and it turned out to be lower using a new path – so the parent was switched and the G and F scores were recalculated. While this change doesn’t seem too important in this example, there are plenty of possible situations where this constant checking will make all the difference in determining the best path to your target.

So how do we determine the path? Simple, just start at the red target square, and work backwards moving from one square to its parent, following the arrows. This will eventually take you back to the starting square, and that’s your path. It should look like the following illustration. Moving from the starting square A to the destination square B is simply a matter of moving from the center of each square (the node) to the center of the next square on the path, until you reach the target.

    
[Figure 7]

# Summary of the A\* Method

Okay, now that you have gone through the explanation, let’s lay out the step-by-step method all in one place:

1) Add the starting square (or node) to the open list.

2) Repeat the following:

a) Look for the lowest F cost square on the open list. We refer to this as the current square.

b) Switch it to the closed list.

c) For each of the 8 squares adjacent to this current square …

* If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.
* If it isn’t on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
* If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.

d) Stop when you:

* Add the target square to the closed list, in which case the path has been found (see note below), or
* Fail to find the target square, and the open list is empty. In this case, there is no path.

3) Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

Note: In earlier versions of this article, it was suggested that you can stop when the target square (or node) has been added to the open list, rather than the closed list.  Doing this will be faster and it will almost always give you the shortest path, but not always.  Situations where doing this could make a difference are when the movement cost to move from the second to the last node to the last (target) node can vary significantly -- as in the case of a river crossing between two nodes, for example.

**Notes on Implementation**

Now that you understand the basic method, here are some additional things to think about when you are writing your own program. Some of the following materials reference the program I wrote in C++ and Blitz Basic, but the points are equally valid in other languages.

***1.  Other Units (collision avoidance):*** If you happen to look closely at my example code, you will notice that it completely ignores other units on the screen. The units pass right through each other. Depending on the game, this may be acceptable or it may not. If you want to consider other units in the pathfinding algorithm and have them move around one another, I suggest that you only consider units that are either stopped or adjacent to the pathfinding unit at the time the path is calculated, treating their current locations as unwalkable. For adjacent units that are moving, you can discourage collisions by penalizing nodes that lie along their respective paths, thereby encouraging the pathfinding unit to find an alternate route (described more under #2).

If you choose to consider other units that are moving and not adjacent to the pathfinding unit, you will need to develop a method for predicting where they will be at any given point in time so that they can be dodged properly. Otherwise you will probably end up with strange paths where units zig-zag to avoid other units that aren't there anymore.

You will also, of course, need to develop some collision detection code because no matter how good the path is at the time it is calculated, things can change over time. When a collision occurs a unit must either calculate a new path or, if the other unit is moving and it is not a head-on collision, wait for the other unit to step out of the way before proceeding with the current path.

These tips are probably enough to get you started. If you want to learn more, here are some links that you might find helpful:

* [Steering Behavior for Autonomous Characters](http://www.red3d.com/cwr/steer/): Craig Reynold's work on steering is a bit different from pathfinding, but it can be integrated with pathfinding to make a more complete movement and collision avoidance system.
* [The Long and Short of Steering in Computer Games](http://ducati.doc.ntu.ac.uk/uksim/uksim%2704/Papers/Simon%20Tomlinson-%2004-20/paper04-20%20CR.pdf): An interesting survey of the literature on steering and pathfinding. This is a pdf file.
* [Coordinated Unit Movement](http://www.gamasutra.com/features/game_design/19990122/movement_01.htm): First in a two-part series of articles on formation and group-based movement by Age of Empires designer Dave Pottinger.
* [Implementing Coordinated Movement](http://www.gamasutra.com/features/19990129/implementing_01.htm): Second in Dave Pottinger's two-part series.

***2. Variable Terrain Cost:***In this tutorial and my accompanying program, terrain is just one of two things – walkable or unwalkable. But what if you have terrain that is walkable, but at a higher movement cost? Swamps, hills, stairs in a dungeon, etc. – these are all examples of terrain that is walkable, but at a higher cost than flat, open ground. Similarly, a road might have a lower movement cost than the surrounding terrain.

This problem is easily handled by adding the terrain cost in when you are calculating the G cost of any given node. Simply add a bonus cost to such nodes. The A\* pathfinding algorithm is already written to find the lowest cost path and should handle this easily. In the simple example I described, when terrain is only walkable or unwalkable, A\* will look for the shortest, most direct path. But in a variable-cost terrain environment, the least cost path might involve traveling a longer distance – like taking a road around a swamp rather than plowing straight through it.

An interesting additional consideration is something the professionals call “influence mapping.” Just as with the variable terrain costs described above, you could create an additional point system and apply it to paths for AI purposes. Imagine that you have a map with a bunch of units defending a pass through a mountain region. Every time the computer sends somebody on a path through that pass, it gets whacked. If you wanted, you could create an influence map that penalized nodes where lots of carnage is taking place. This would teach the computer to favor safer paths, and help it avoid dumb situations where it keeps sending troops through a particular path, just because it is shorter (but also more dangerous).

Yet another possible use is penalizing nodes that lie along the paths of nearby moving units. One of the downsides of A\* is that when a group of units all try to find paths to a similar location, there is usually a significant amount of overlap, as one or more units try to take the same or similar routes to their destinations. Adding a penalty to nodes already 'claimed' by other units will help ensure a degree of separation, and reduce collisions. Don't treat such nodes as unwalkable, however, because you still want multiple units to be able to squeeze through tight passageways in single file, if necessary. Also, you should only penalize the paths of units that are near the pathfinding unit, not all paths, or you will get strange dodging behavior as units avoid paths of units that are nowhere near them at the time. Also, you should only penalize path nodes that lie along the current and future portion of a path, not previous path nodes that have already been visited and left behind.

***3. Handling Unexplored Areas:***Have you ever played a PC game where the computer always knows exactly what path to take, even though the map hasn't been explored yet? Depending upon the game, pathfinding that is too good can be unrealistic. Fortunately, this is a problem that is can be handled fairly easily.

The answer is to create a separate "knownWalkability" array for each of the various players and computer opponents (each player, not each unit -- that would require a lot more computer memory). Each array would contain information about the areas that the player has explored, with the rest of the map assumed to be walkable until proven otherwise. Using this approach, units will wander down dead ends and make similar wrong choices until they have learned their way around. Once the map is explored, however, pathfinding would work normally.

***4. Smoother Paths:***While A\* will automatically give you the shortest, lowest cost path, it won't automatically give you the smoothest looking path. Take a look at the final path calculated in our example (in Figure 7). On that path, the very first step is below, and to the right of the starting square. Wouldn't our path be smoother if the first step was instead the square directly below the starting square?

There are several ways to address this problem. While you are calculating the path you could penalize nodes where there is a change of direction, adding a penalty to their G scores. Alternatively, you could run through your path after it is calculated, looking for places where choosing an adjacent node would give you a path that looks better. For more on the whole issue, check out [Toward More Realistic Pathfinding](http://www.gamasutra.com/features/20010314/pinter_01.htm), a (free, but registration required) article at Gamasutra.com by Marco Pinter.

***5. Non-square Search Areas:***In our example, we used a simple 2D square layout. You don't need to use this approach. You could use irregularly shaped areas. Think of the board game Risk, and the countries in that game. You could devise a pathfinding scenario for a game like that. To do this, you would need to create a table for storing which countries are adjacent to which, and a G cost associated with moving from one country to the next. You would also need to come up with a method for estimating H. Everything else would be handled the same as in the above example. Instead of using adjacent squares, you would simply look up the adjacent countries in the table when adding new items to your open list.

Similarly, you could create a waypoint system for paths on a fixed terrain map. Waypoints are commonly traversed points on a path, perhaps on a road or key tunnel in a dungeon. As the game designer, you could pre-assign these waypoints. Two waypoints would be considered "adjacent" to one another if there were no obstacles on the direct line path between them. As in the Risk example, you would save this adjacency information in a lookup table of some kind and use it when generating your new open list items. You would then record the associated G costs (perhaps by using the direct line distance between the nodes) and H costs (perhaps using a direct line distance from the node to the goal). Everything else would proceed as usual.

Amit Patel has written a brief [article](http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html) delving into some alternatives. For another example of searching on an isometric RPG map using a non-square search area, check out my article [Two-Tiered A\* Pathfinding](http://www.policyalmanac.org/games/twoTiered.htm).

***6. Some Speed Tips:***As you develop your own A\* program, or adapt the one I wrote, you will eventually find that pathfinding is using a hefty chunk of your CPU time, particularly if you have a decent number of pathfinding units on the board and a reasonably large map. If you read the stuff on the net, you will find that this is true even for the professionals who design games like Starcraft or Age of Empires. If you see things start to slow down due to pathfinding, here are some ideas that may speed things up:

* Consider a smaller map or fewer units.
* Never do path finding for more than a few units at a time. Instead put them in a queue and spread them out over several game cycles. If your game is running at, say, 40 cycles per second, no one will ever notice. But they will notice if the game seems to slow down every once in a while when a bunch of units are all calculating paths at the same time.
* Consider using larger squares (or whatever shape you are using) for your map. This reduces the total number of nodes searched to find the path. If you are ambitious, you can devise two or more pathfinding systems that are used in different situations, depending upon the length of the path. This is what the professionals do, using large areas for long paths, and then switching to finer searches using smaller squares/areas when you get close to the target. If you are interested in this concept, check out my article [Two-Tiered A\* Pathfinding](http://www.policyalmanac.org/games/twoTiered.htm).
* For longer paths, consider devising precalculated paths that are hardwired into the game.
* Consider pre-processing your map to figure out what areas are inaccessible from the rest of the map. I call these areas "islands." In reality, they can be islands or any other area that is otherwise walled off and inaccessible. One of the downsides of A\* is that if you tell it to look for paths to such areas, it will search the whole map, stopping only when every accessible square/node has been processed through the open and closed lists. That can waste a lot of CPU time. It can be prevented by predetermining which areas are inaccessible (via a flood-fill or similar routine), recording that information in an array of some kind, and then checking it before beginning a path search.
* In a crowded, maze-like environment, consider tagging nodes that don't lead anywhere as dead ends. Such areas can be manually pre-designated in your map editor or, if you are ambitious, you could develop an algorithm to identify such areas automatically. Any collection of nodes in a given dead end area could be given a unique identifying number. Then you could safely ignore all dead ends when pathfinding, pausing only to consider nodes in a dead end area if the starting location or destination happen to be in the particular dead end area in question.

***7. Maintaining the Open List:*** This is actually one of the most time consuming elements of the A\* pathfinding algorithm. Every time you access the open list, you need to find the square that has the lowest F cost. There are several ways you could do this. You could save the path items as needed, and simply go through the whole list each time you need to find the lowest F cost square. This is simple, but really slow for long paths. This can be improved by maintaining a sorted list and simply grabbing the first item off the list every time you need the lowest F-cost square. When I wrote my program, this was the first method I used.

This will work reasonably well for small maps, but it isn’t the fastest solution. Serious A\* programmers who want real speed use something called a binary heap, and this is what I use in my code. In my experience, this approach will be at least 2-3 times as fast in most situations, and geometrically faster (10+ times as fast) on longer paths. If you are motivated to find out more about binary heaps, check out my article, [Using Binary Heaps in A\* Pathfinding](http://www.policyalmanac.org/games/binaryHeaps.htm).

Another possible bottleneck is the way you clear and maintain your data structures between pathfinding calls. I personally prefer to store everything in arrays. While nodes can be generated, recorded and maintained in a dynamic, object-oriented manner, I find that the amount of time needed to create and delete such objects adds an extra, unnecessary level of overhead that slows things down. If you use arrays, however, you will need to clean things up between calls. The last thing you will want to do in such cases is spend time zero-ing everything out after a pathfinding call, especially if you have a large map.

I avoid this overhead by creating a 2d array called whichList(x,y) that designates each node on my map as either on the open list or closed list. After pathfinding attempts, I do not zero out this array. Instead I reset the values of onClosedList and onOpenList in every pathfinding call, incrementing both by +5 or something similar on each path finding attempt. This way, the algorithm can safely ignore as garbage any data left over from previous pathfinding attempts.I also store values like F, G and H costs in arrays. In this case, I simply write over any pre-existing values and don't bother clearing the arrays when I'm done.

Storing data in multiple arrays consumes more memory, though, so there is a trade off. Ultimately, you should use whatever method you are most comfortable with.

***8. Dijkstra's Algorithm:***While A\* is generally considered to be the best pathfinding algorithm (see rant above), there is at least one other algorithm that has its uses - Dijkstra's algorithm. Dijkstra's is essentially the same as A\*, except there is no heuristic (H is always 0). Because it has no heuristic, it searches by expanding out equally in every direction. As you might imagine, because of this Dijkstra's usually ends up exploring a much larger area before the target is found. This generally makes it slower than A\*.

So why use it? Sometimes we don't know where our target destination is. Say you have a resource-gathering unit that needs to go get some resources of some kind. It may know where several resource areas are, but it wants to go to the closest one. Here, Dijkstra's is better than A\* because we don't know which one is closest. Our only alternative is to repeatedly use A\* to find the distance to each one, and then choose that path. There are probably countless similar situations where we know the kind of location we might be searching for, want to find the closest one, but not know where it is or which one might be closest.

**Further Reading**

Okay, now you have the basics and a sense of some of the advanced concepts. At this point, I’d suggest wading into my source code. The package contains two versions, one in C++ and one in Blitz Basic. Both versions are heavily commented and should be fairly easy to follow, relatively speaking.  Here is the link.

* [Sample Code: A\* Pathfinder (2D) Version 1.9](http://www.policyalmanac.org/games/AStar.zip)

If you do not have access to C++ or Blitz Basic, two small exe files can be found in the C++ version. The Blitz Basic version can be run by downloading the free demo version of Blitz Basic 3D (not Blitz Plus) at the [Blitz Basic](http://www.blitzbasic.com/) web site.

You should also consider reading through the following web pages. They should be much easier to understand now that you have read this tutorial.

* [Amit’s A\* Pages](http://www-cs-students.stanford.edu/~amitp/gameprog.html#Paths): This is a very widely referenced page by Amit Patel, but it can be a bit confusing if you haven’t read this article first. Well worth checking out. See especially Amit's own [thoughts](http://theory.stanford.edu/~amitp/GameProgramming/) on the topic.
* [Smart Moves: Intelligent Path Finding](http://www.gamasutra.com/features/19970801/pathfinding.htm): This article by Bryan Stout at Gamasutra.com requires registration to read. The registration is free and well worth it just to reach this article, much less the other resources that are available there. The program written in Delphi by Bryan helped me learn A\*, and it is the inspiration behind my A\* program. It also describes some alternatives to A\*.
* [Terrain Analysis](http://www.gamasutra.com/features/gdcarchive/2000/pottinger.doc): This is an advanced, but interesting, article by Dave Pottinger, a professional at Ensemble Studios. This guy coordinated the development of Age of Empires and Age of Kings. Don’t expect to understand everything here, but it is an interesting article that might give you some ideas of your own. It includes some discussion of mip-mapping, influence mapping, and some other advanced AI/pathfinding concepts.

Some other sites worth checking out:

* [aiGuru: Pathfinding](http://www.aiguru.com/pathfinding.htm)
* [Game AI Resource: Pathfinding](http://www.gameai.com/pathfinding.html)
* [GameDev.net: Pathfinding](http://www.gamedev.net/reference/list.asp?categoryid=18#94)

I also highly recommend the following books, which have a bunch of articles on pathfinding and other AI topics. They also have CDs with sample code. I own them both. Plus, if you buy them from Amazon through these links, I'll get a few pennies from Amazon. :)

Well, that’s it. If you happen to write a program that uses any of these concepts, I’d love to see it. I can be reached at

http://www.policyalmanac.org/graphics/mail2.jpg

Until then, good luck!

# Games and adversarial search

In algorithms so far, we've imagined the world through the eyes of a single agent. It was a single agent searching for the shortest path between two places, without interference from another agent that might have an agenda of its own.

Adversarial search algorithms look for the best solution when something can go wrong, or when there is another agent involved in the search who could have conflicting objectives, i.e. there is another agent serving as the adversary.

## Games

The need for a search algorithm that considers the actions of an adversary shows up in games. There can be single-player games, where you have

* perfect information of the game
* know the rules of the game
* know the state that your actions produce
* know when you win

Examples of single player games are the rubiks cube, or the 8-tile puzzle. For these types of games, using a depth-limited search to build a search tree for your next few moves could provide you with complete information about the game state for your next few moves. There's no adversary in the game to mess with you.

Games with an adversary are generally what we think of when we think of games. In these games, there is another player, or players, with motives alterior to ours. Games such as checkers, chess, and othello are examples.

### Game features

**Deterministic or stochastic?**

Deterministic: no uncertainty, actions produce same results.

Stochastic: uncertainty, e.g. games with dice.

**Number of players**

One, two, or multi-player

**Zero-sum?**

Zero-sum: One utility function is shared between players, and one player's gain in utility is offset by another player's loss in utility. The game has one winner and one loser.

**Perfect information**

Players can see the entire board. In chess, players have perfect information. In poker, not perfect information.

### History of computers and playing games

**Checkers:** Computers started winning in 1994. Two players with perfect play will draw.

**Chess:** Deep Blue beat Gary Kasparov in 1997.

**Othello:** Humans don't beat computers playing Othello

**Go:** Big Othello, too many combinations for a computer. Human still have a chance.

## Adversarial search

We want a search algorithm that determines the best strategy and recommends a move for each state in the game. This is equivalent to determining the set of actions needed to achieve your goals, such as winning the game. (Just like the other search algorithms we learned.) In adversarial games, the algorithm needs to "reason" ahead to determine the best actions considering both your goals and the actions of your adversary.

Setting up the search problem

If all we had to do was find the maximum path through the tree, that would be trivial. What makes the game interesting is adding in the actions of an adversary. For every move that one player makes to maximize their own chance of winning, the adversary makes a counter move for their own benefit. The game state after the adversary's move might not have a path to winning for the other player.

The game can be defined as a search problem with the following:

S0 : initial state of the game

Player(s): defines which player has the move in state s.

Actions(s): set of legal moves in state s.

Result(s, a): defines the result of an action a in state s.

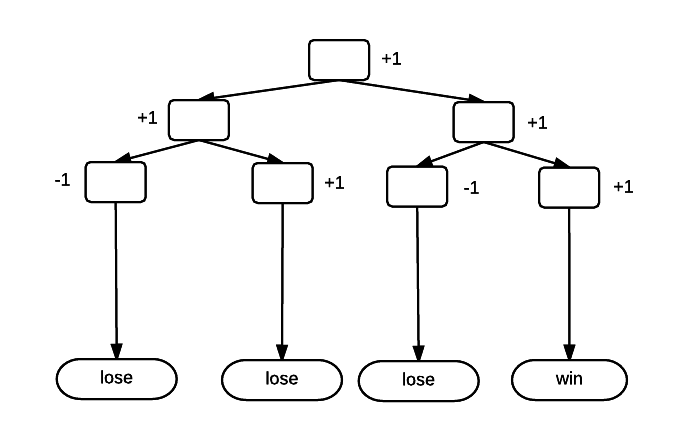
Terminal-test(s): true when the game is over, false otherwise. States at the end of the game are terminal states.

Utility(s, p): defines the numeric value for game that ends in terminal state s for player p. Also called the objective function. In some games, such as chess, the outcome is win, lose, or draw. In other games, such as gin rummy, the utility could be a point total within a specified range.

The search tree consists of game states, where nodes in the tree are determined by actions in the game and specifics of the tree depend on the rules of the game. The solution that the algorithm provides maps states to actions.

**Example: Two-player, zero-sum, perfect information game**

Each node in the tree is a state in the game, and from the node, there is a best-possible outcome to the game, win or lose. Let +1 be winning the game and -1 be losing the game. From the root, the best possible outcome is winning, which means there is a path to winning from the root. There is also a path to losing. As the tree is built, the best possible outcome from that state in the tree is maintained. If the value assigned to the node is a +1, it means there is still a path to winning from that state.



The value assigned to any node in the tree is the maximum value of its children.

Assume there are two players in the game: MAX and MIN. MAX's goal is to get the maximum score, and MIN's goal is to minimize MAX's score. Each player, being a rational agent, will select the move that provides it with the greatest utility.

The optimal move for each player can be calculated using the Minimax algorithm.

**Minimax algorithm**

The minimax algorithm requires a recursive traversal of the game tree. Starting at the root, MAX selects the action that returns the maximum of the minimum values of its children. The algorithm begins with a call to minimax(), with the node to evaluate, the current depth, and whether it's MAX or MIN making the decision as arguments.

minimax(node, depth, maxPlayer)

if depth == 0 or terminal(node) //terminal test is true

return f(node) //evaluation of the node

if maxPlayer //Player(s) = MAX

bestValue = -MAX\_INT //system property, maximum negative integer

for each child in node.adjacent

eval = minimax(n, depth - 1, FALSE)

bestValue = max(bestValue, eval)

return bestValue

else //Player(s) = MIN

bestValue = MAX\_INT

for each child in node.adjacent

eval = minimax(child, depth - 1, TRUE)

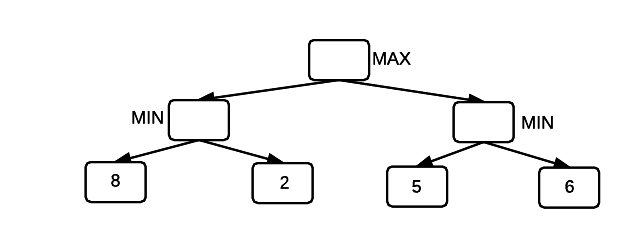
bestValue = min(bestValue, eval)

return bestValue

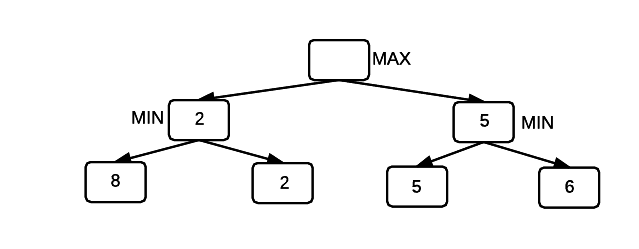
minimax(origin, depth, TRUE) //call from root for MAX player

## Minimax Search

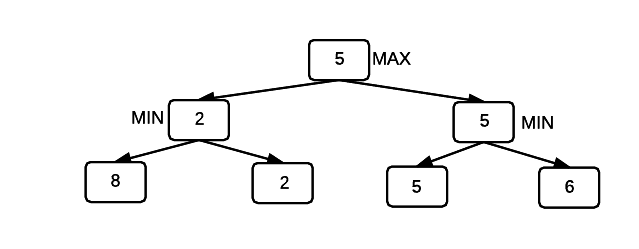
For this search, we start with a game tree of all possible futures. At the bottom of the tree, or the bottom level being explored, are the outcomes for the game and the utility for that outcome. Assume a simple game tree that looks like the following:



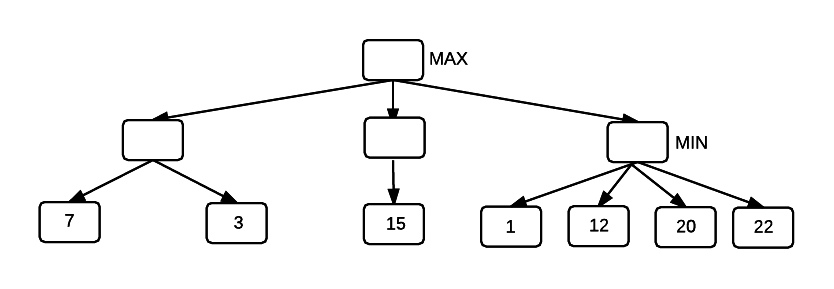
In this example, the utility is not just 0 or 1 or win or lose, but rather a numerical value within a range of values. Each level in the tree belongs to either MAX or MIN, which means that MAX or MIN decides which of the values from the next level to select. At the root of the tree, MAX makes the decision about which path to follow from its two children. (This example has two children, but it's not a requirement that there are only two options). But, MAX doesn't know the utility associated with either decision. The next decision belongs to MIN, and in this example, MIN "sees" the utility of each decision in its child nodes. MIN chooses the node with the minimum utility. In this case, it's the 2 and 5. These values propagate up one level, and the tree now looks like:



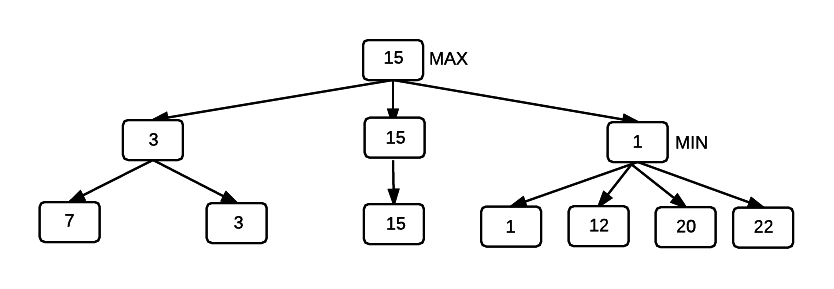
Next, MAX chooses the maximum value from its two options, which is the 5. The game ends with a utility of 5 for MAX. For MAX, the minimax algorithm will maximize its minimum gain.



**Example: Use the minimax algorithm to find the utility for MAX in the following game tree.**



Recurse to the bottom of the tree, and return the minimum value for each node. At the next level up in the tree, return the maximum of the minimum values. The utility to MAX is 15.



**Evaluating non-terminal nodes**

In any real game, calculating all possible outcomes to the game isn't really an option, and that's what would be required in order for the "recurse to bottom of tree" step to work. Instead, a depth-limited search is employed, and the non-terminal nodes are scored as if they were terminal. The question is, how do we score a node when we don't know the outcome? The score is an estimate of the utility of the state. An evaluation function is used to calculate the utility value to a game state. The evaluation function includes features of the game that provide evidence about the utility of the game state. These features are pre-determined based on expertise of the game.

**Example: Evaluating the game state in a chess**

There are several features that could indicate who is on the best path to winning in a game of chess.

* Queen advantage
* Pawn advantage
* Control of the board
* Differential in number of pieces overall
* Using these state variables, a weight a applied to each variable to built the evaluation function:

eval(s) = w1f1(s) + w2f2(s) + ... + wnfn(s)

The weight of each features determines now much that feature is worth using input from domain experts and/or a learning system that determines the weights.

An example of how the control the board feature could be set up is:

fx(s) = (squares controlled by black player - squares controlled by white player)

The state then gets a score, such as 0.7. States with a higher score are more desirable than states with a lower score.

# Minimax with alpha beta pruning

The minimax algorithm finds the optimal move in a 2-player game. However, it's too computationally intensive in a real game to search the entire game tree.

Depth-limited search and an evaluation function that estimates the utility of a game state can be used. The evaluation function provides a heuristic for estimating the utility of the game state. The search is limited to a fixed depth, and once that depth is reached, the path that leads to the highest evaluation value is selected. A deeper search will yield better results, but will also require more time. As with any depth-limited search, the critical move might be one level deeper in the tree, just over the horizon of the search.

The runtime complexity of minimax is bd, where b is the branching factor and d is the search depth.

In any search tree, there are branches in the tree that cannot possibly contain the optimal strategy for the game, and it is possible to detect these branches and prune them so that time is not wasted on searching them. Pruning useless branches from the search tree makes minimax much faster, in some cases, exponentially faster.

The alpha-beta algorithm finds the optimal minimax solution while avoiding subtrees that won't contain the optimal solution. Alpha-beta pruning sets two bounds on values observed during computation.

**MAX:**

At the MAX node, store a value, called alpha (α), which is a lower bound on the exact minimax score. With optimal play MAX can score at least α, and might score better. If MIN chooses nodes with score < α, then MAX can't score better than α. This value can also be thought of as the maximum lower bound on the game outcome.

**MIN:**

At the MIN node, store a value, called beta (β), which is an upper bound on the exact minimax score. With optimal play MIN can score no more than β, and might score less.

During the search process, alpha values can never decrease and beta values can never increase. The search can be discontinued at a node, i.e. the branch below the node can be pruned, if:

It is a MAX node and the alpha value is ≥ the beta of any MIN ancestor (known as a beta cutoff). An ancestor is a node further up in the search tree.

It is a MIN node and the beta value is ≤ the alpha of any MAX ancestor (known as an alpha cutoff).

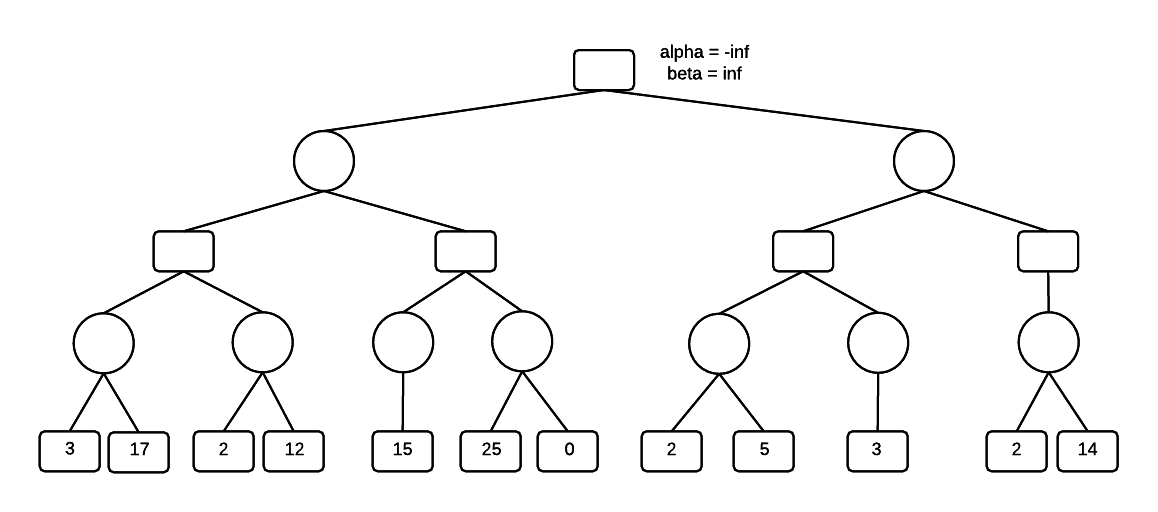
Nodes can only be part of the solution if alpha <= N <= beta, where N is the current estimate of the node's value. If N is outside of the range, it could never result in the solution path and the branch can be pruned. The algorithm skips processing the children and moves back to the parent node.

**Example: Demonstrate the AB algorithm on the following tree.**

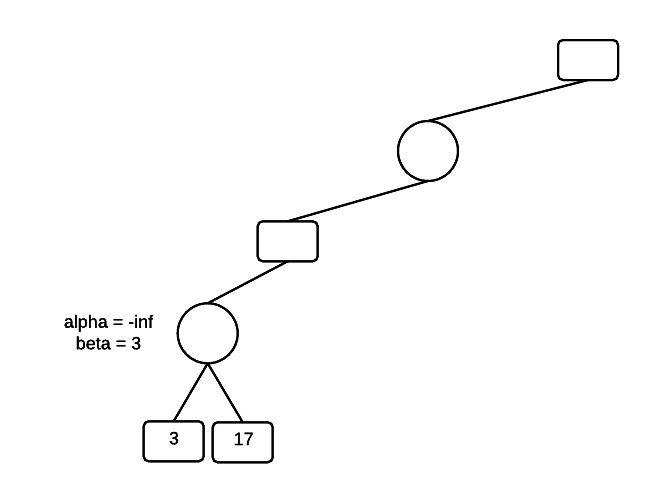
Square nodes are MAX.

Circle nodes are MIN.

Start at the root, initialize alpha to -infinity and beta to inf.

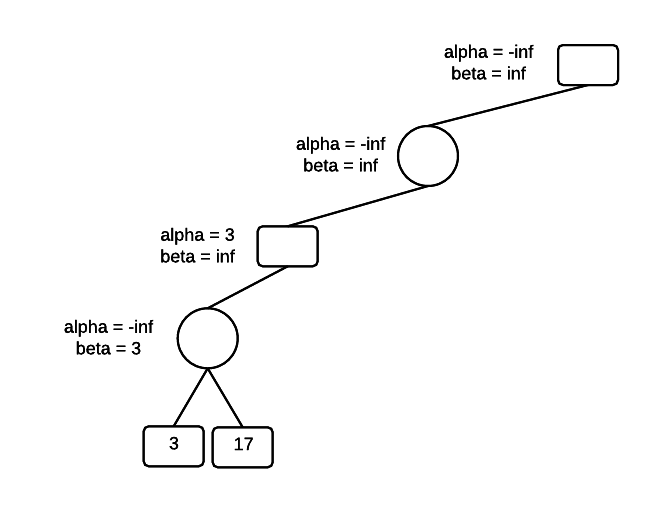


Move to the bottom of the tree, passing the values of alpha and beta all the way down. The first player to select is MIN, and MIN will select the minimum of its terminal-node children, which is the 3. Set beta = 3 and leave alpha unchanged.

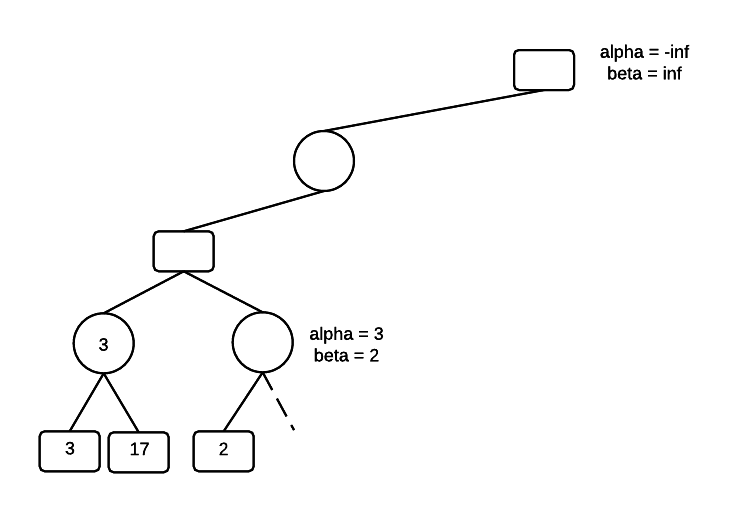


Next, evaluate the other child of MIN, which is the 17. Since 17 > 3, MIN won't select the 17 and the value of beta is unchanged.

Once we have a MIN value, we know that MAX will be at least 3 and we can set the minimum for MAX, alpha = 3. We've moved up a level in the tree and there was an existing value for beta at this level. That value is left unchanged.

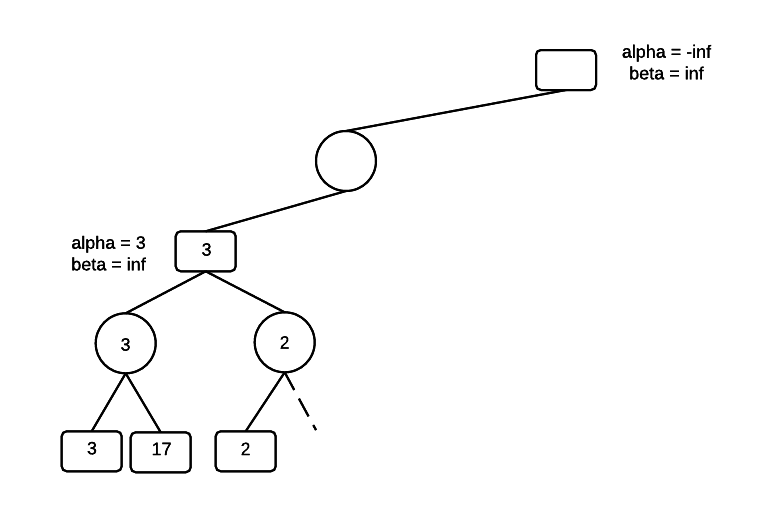


Next, go to the next child for MAX and traverse to the bottom of the tree. The first node that MIN evaluates has a value of 2, which sets beta = 2.



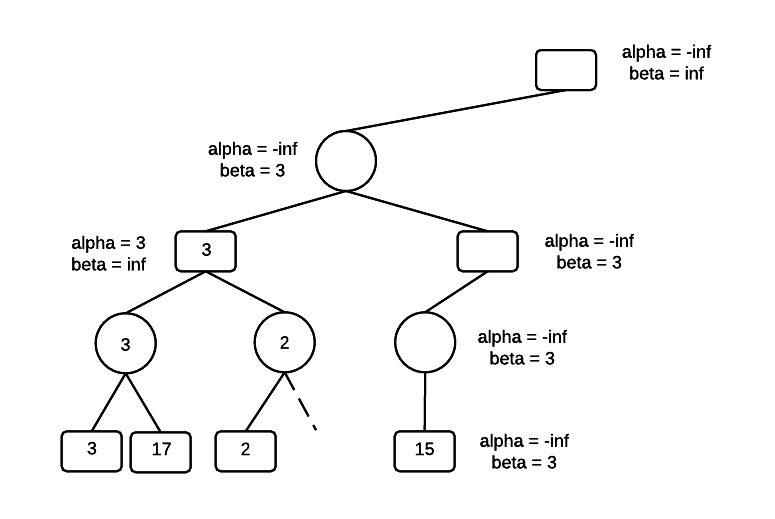
MIN now has a maximum value of 2. If its other children have values higher than 2, MIN won't select them. However, MAX has a minimum value of 3 in its other child, so MAX won't select the 2. We can prune MINs other children and not evaluate them. We also see that the condition alpha <= N <= beta is violated.

Set the MAX value.

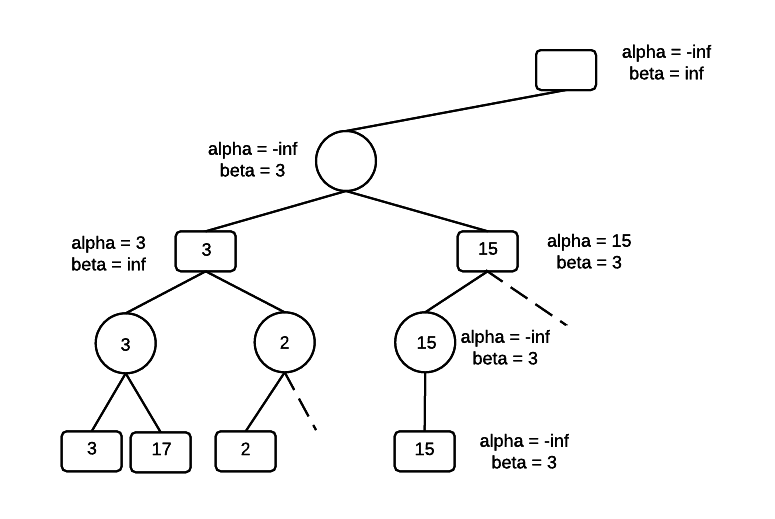


Move to the MIN parent node and set beta = 3. MIN might find a value lower than 3 in the other branch of the tree, but it will not select a value greater than 3. MIN will do no worse than 3.

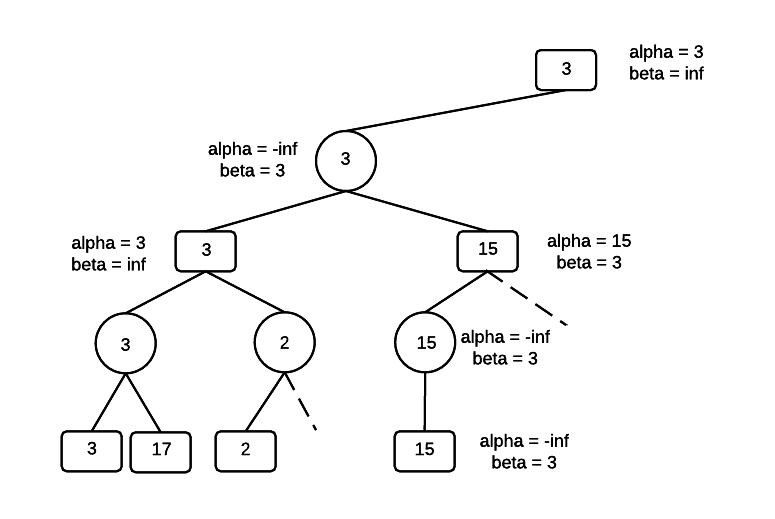
Recurse to the bottom of the tree, carrying the current alpha and beta values at each level. At the bottom of the tree, MIN encounters a value of 15. However, beta is not changed because one of the rules of alpha-beta pruning is that beta never increases.



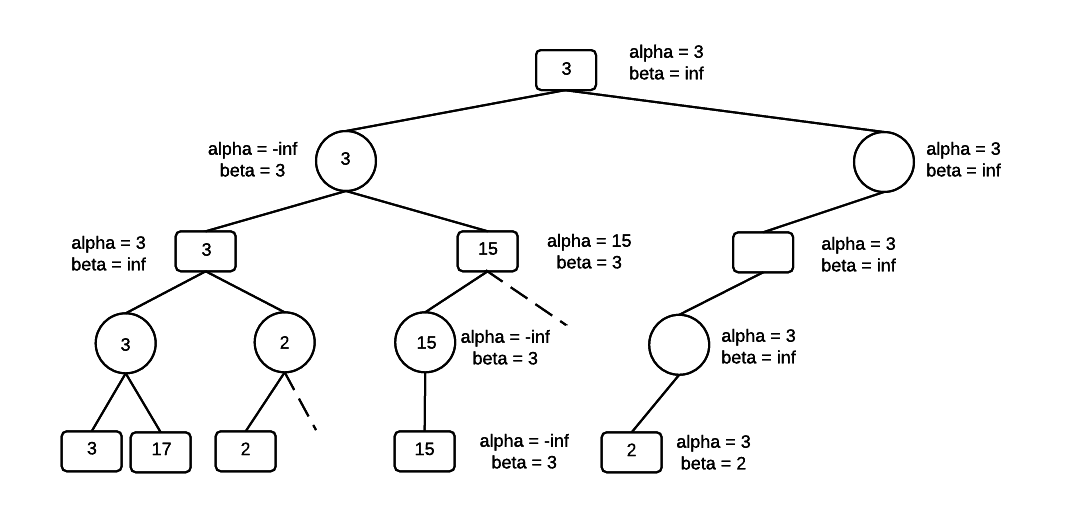
At the MAX level, alpha is set to 15. We now have a violation of the constraint that alpha <= N <= beta. Also, searching MAXs other branch is unnecessary because even if MAX finds a value greater than 15, its MIN parent will never select it. So, the other branch can be pruned.



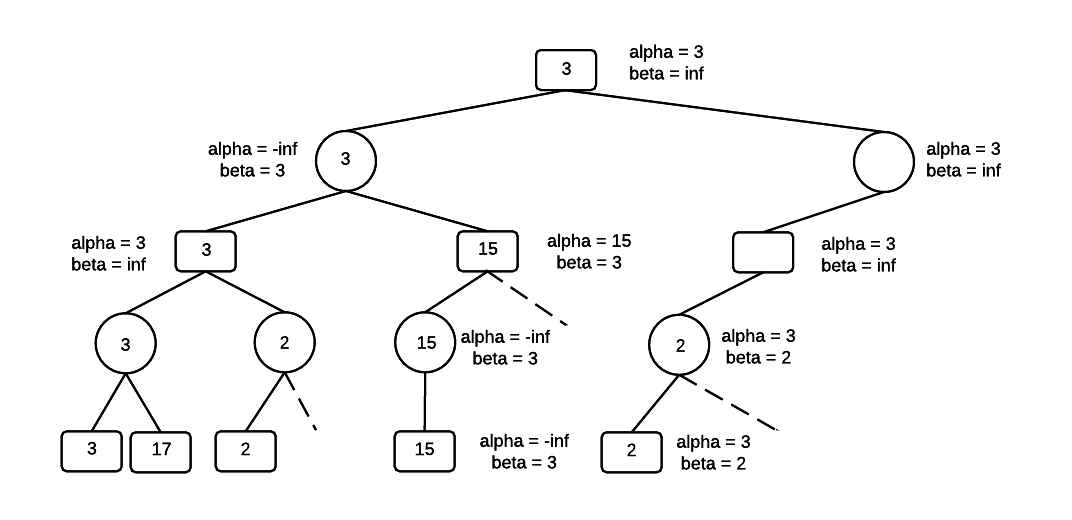
Assign the parent, which is a MIN node. Return to the root and set the value for alpha. The root is a MAX node, setting alpha = 3 means that MAX gets an outcome that is at least 3.



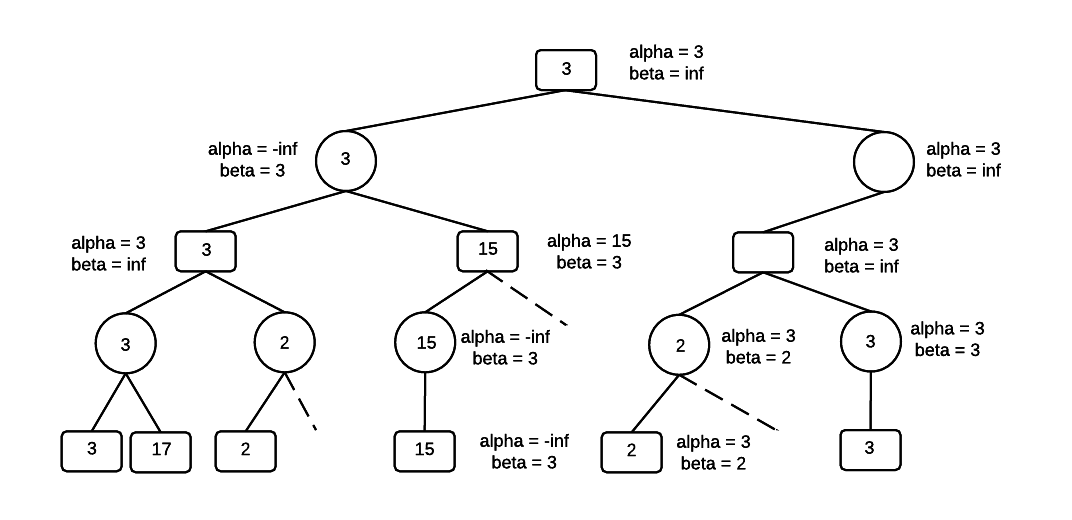
Traverse to the bottom of the tree, along the right branch of the root, carrying the values for alpha and beta. At the bottom of the tree is a value of 2 for MIN, which sets beta = 2. There is now an alpha-beta violation and the other children of MIN can be pruned. MIN won't select a value greater than 2 and if the 2 made it's way up the tree, it would never be selected by MAX at the root.



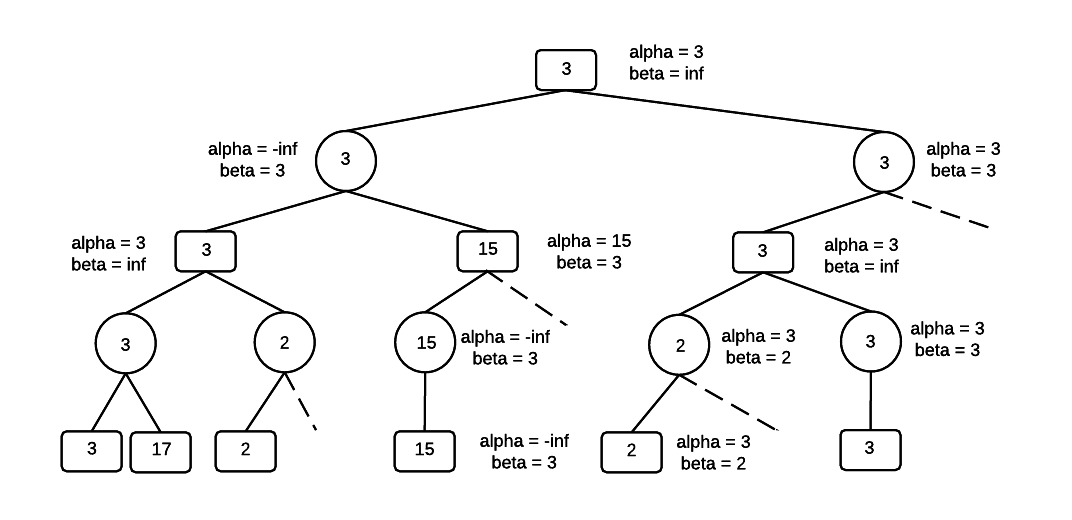
Set the MIN value as 2 and move up to the MAX parent level. Beta is now 2 and alpha is still 3.



From the MAX level, recurse to the bottom of the tree along the right branch, passing the values for alpha and beta all the way to the bottom of the tree. alpha = 3 and beta = inf. The value for MIN is updated using the value of the left-most node, which is 3. Beta is now 3, and alpha is 3.



Move up one level in the tree to the MIN node and set beta = 3. At this point in the algorithm, alpha = beta = 3. The remainder of the tree can be pruned because MIN won't select a value greater than 3, and MAX at the root, won't select a value less than 3. The search completes with a minimax value of 3.



**Maximum Expected Utility (MEU)**

The utility of a state is a real number that captures how well the state represents the agent's goals. The utility tells us how good the state is. It can be difficult to wrap up all of an agent's goals into one number, especially when goals can be conflicting and not necessarily good or bad. The utility can be a continuous measure of quality, not just +/-1.

In real-world situations, there is generally a measure of uncertainty. The utility of a state can include the probability of an outcome to represent an environment where we don't have complete knowledge. The probability of an outcome is conditioned on the evidence and a particular action.

**Result(a):** Result of action a is a random variable, which is a variable with a value subject to variation due to chance. Can take on different values, each with a probability.

We will typically use probabilities conditioned on evidence and actions.

**Pr(Result(a) = s | a, e):** Probability of outcome *s*given evidence *e* and action *a* happening.

**Conditional Probability**

**Pr(A | B):** Probability of A given B.

Pr(forecast = "rain" | sky = "cloudy") Probability of rain given that the sky is cloudy.

**Expected Utility**

The expected utility of a state includes both the utility of the state to the agent, U(s), as well as how likely the state is, Pr(s | a, e).

The expected utility of an action, given the evidence is:

EU(a | e) = Σs Pr(Result(a) = s | a, e) U(s)

An action can have multiple outcome states.

Each state has a utility, and EU includes the sum of the utilities for the possible outcome states.

Each state also has a probability that the action will result in that state.

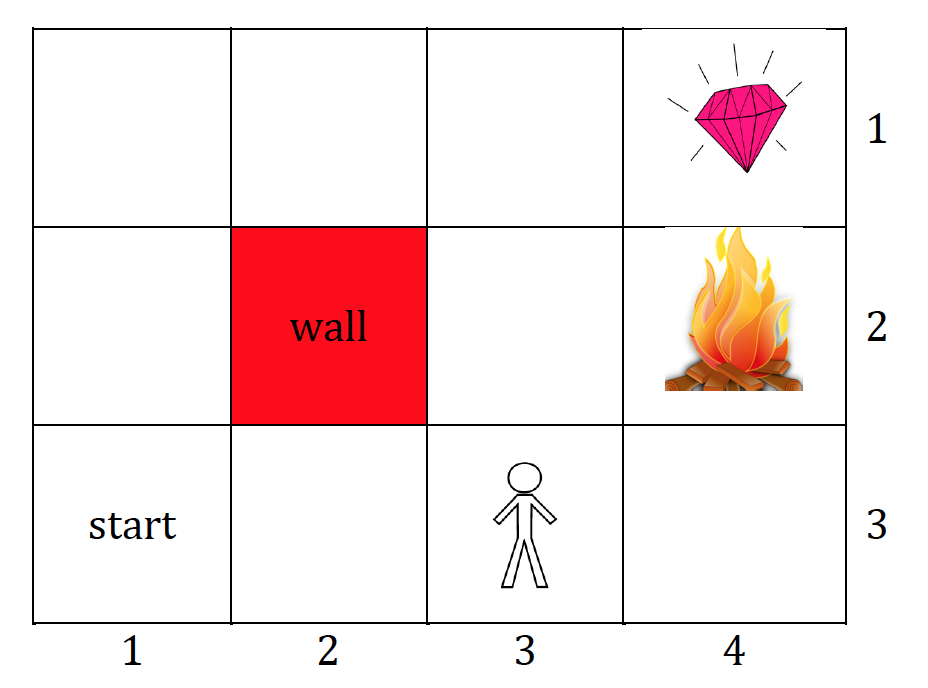
MEU = arga max Σs Pr(Result(a) = s | a, e) U(s)

The agent should choose the action that maximizes its expected utility.

**Markov Decision Processes (MDP)**

In previous examples, a move always worked. When you calculate the utility of a move, you choose the move with the maximum utility and there's no uncertainty about whether that move will succeed. In real-world problems, however, there is uncertainty about whether an action will be successful. Most of the time, when you push on a door, it will open. But, occasionally, it might be locked, or stuck. The fact that your action won't produce the intended result 100% of the time adds an element of uncertainty when calculating the utility of an action.

**Example: Get the agent to the diamond without dying in the fire pit.**



The diamond has a utility of +1 and the firepit has a utility of -1.

With certainty in actions, the solution is trivial, go up two squares and right one square and retrieve the diamond. But, actions are noisy. They don't always go as planned. For any action, there is a probability of success and probabilities for other results. Assume the agent wants to go north. 80% of the time the action is successful, 10% of the time the agent goes east and 10% of the time the agent goes west. If agent goes into a wall, the agent just bounces off and stays in the same location.

**Rewards:**

On each step, there is a small living reward, can be positive or negative. The reward is the utility of the state.

There is big reward at the end, positive or negative.

**Goal:**

Maximize the sum of rewards over time.

**MDP search problems**

An MDP is a non-deterministic, sequential search problem with a Markovian transition model and additive rewards.

**The search problem is defined by:**

A set of states: s ∈ S.

A set of actions: a ∈ A. In the fire pit problem, the actions are Up, Down, Left, Right.

Transition function: T(s, a, s'). The transition function is the same as *Pr(s' | s, a)*, the probability of state *s'* given state *s* and action *a*. Current state is the evidence. For example, if an action is successful 80% of the time, then Pr(s' | s, a), where s' is the intended state for the action a, is .80.

Reward function: R(s, a, s'). The reward for taking action *a* to move from state *s* to state *s'*. There can be a small living reward, and a big + or - reward at the goal.

Solution to an MDP:

A solution for an MDP specifies what the agent should do for any state that the agent might reach. It's not a fixed sequence of actions, but rather an action for a state.

The solution is called a policy, typically denoted as π, where π(s) is the action recommended by policy π in state s.

The quality of a policy is its expected utility, and the optimal policy is the one with the highest expected utility. In calculating the optimal policy, we want to maximize expected utility.

**Markov Assumption**

In an MDP, we are looking for a sequence of actions that solve the problem. To simplify the search, we assume that the probability distribution of future states depends only on the current state, not on any previous states.

Pr(St+1 = s' | St = s, At = a)

The probability of an action being successful and being in a certain state at t+1 depends only on the action and the state at t, not the state at t-1, or t-2, ... and so on.

**MDP Policies**

In MDPs, we want a policy that maps states to actions: π: s -> a.

1. Policy π gives action for each state.

2. The optimal policy maximizes the expected utility if followed.

Having a specified action for each state creates a reflex agent. The agent isn't learning, or revising its plans. But rather, the policy is pre-specified. In any state, the agent always knows what to do next.

**Examples of Optimal Policies**

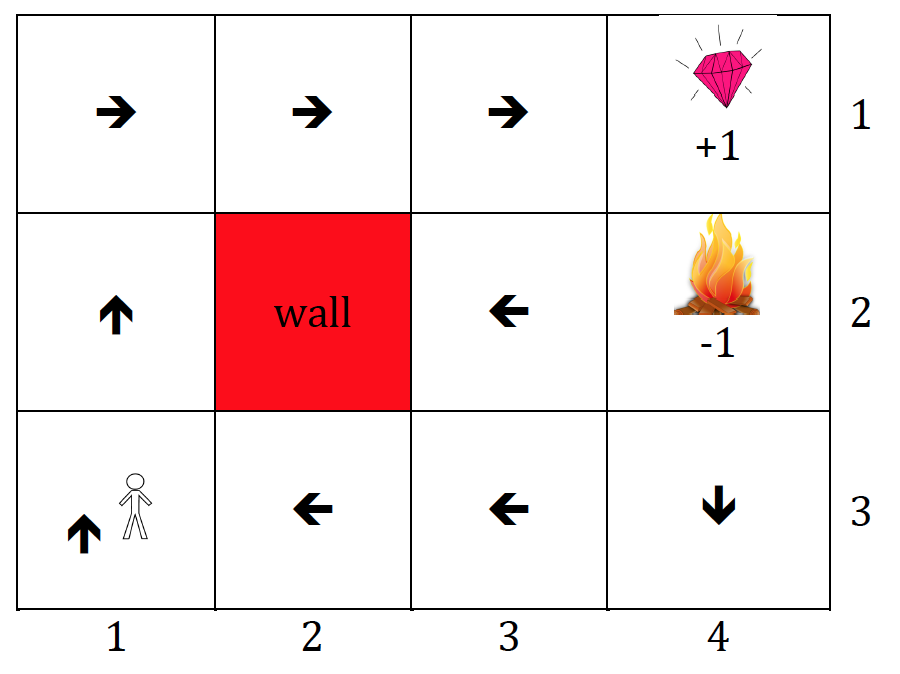
Using world with fire pit and diamond shown above. 80% of time action is successful, and 20% of time action is unsuccessful. From starting location, assume agent wants to go up. 10% of time agent will actually go left and 10% of time agent will go right. If agent wants to go left, it will be successful 80% of time, 10% of time will go up, and 10% of time will go down.

In each of the following examples, the arrow in the square shows the optimal policy for that state.

**Example 1:**

R(s) = -0.01

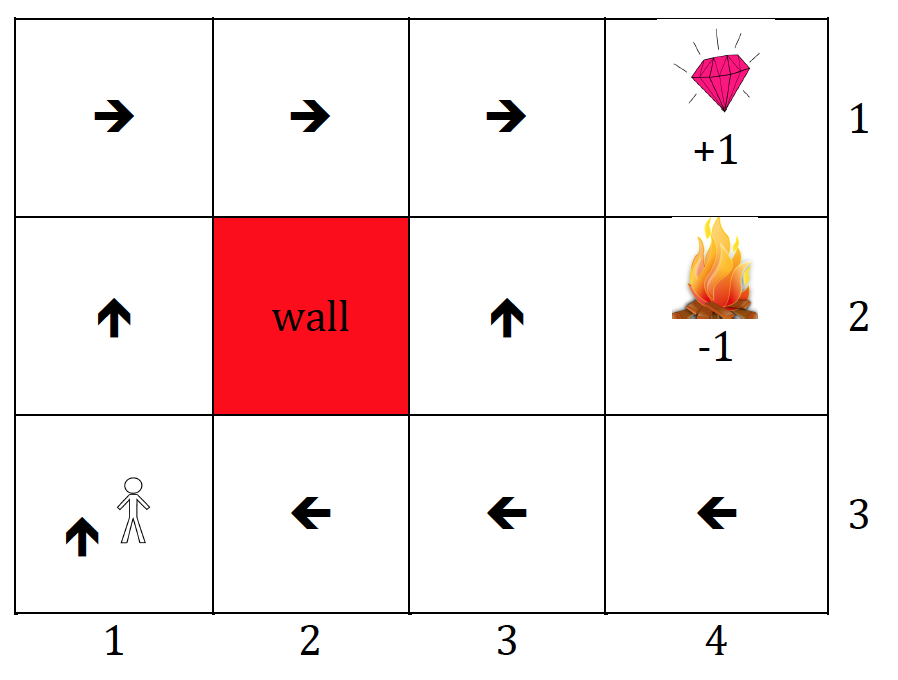
Slightly negative living reward. On each iteration, the living reward is deducted from the agent's utility. The optimal policy involves avoiding the fire pit at all costs. In the square next to the pit, run into the wall to avoid the risk of accidentally going into the pit. The living reward deduction is so small that the agent has time to hit the wall over and over.



**Example 2:**

R(s) = -0.03.

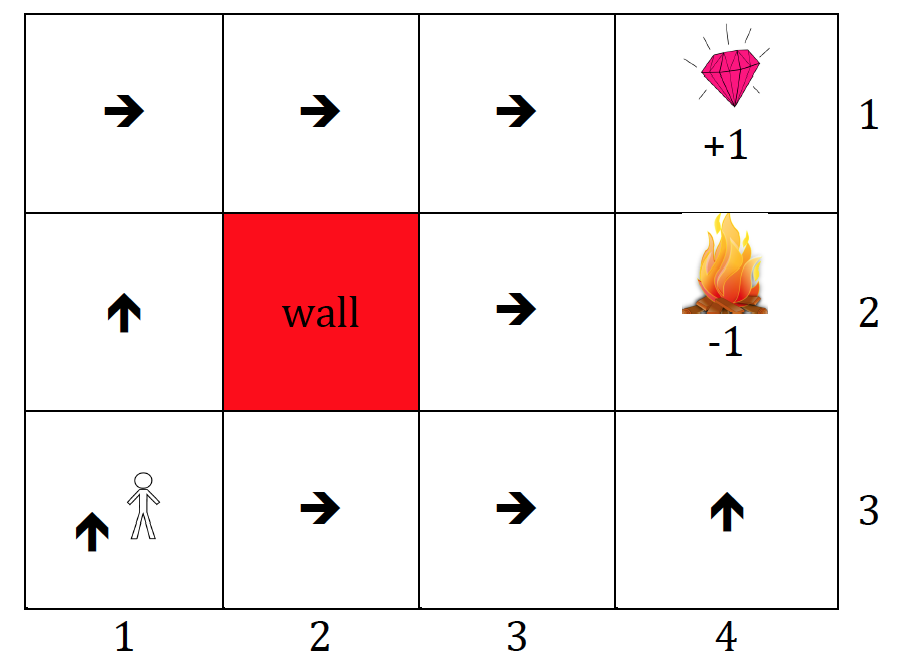
A slightly larger negative living reward changes the agent's behavior a little. It is no longer the best policy to run into the wall. The increased risk of dying in the fire is worth it to speed up the path to the diamond.



**Example 3:**

R(s) = -2.0

The living reward is so bad, the jumping into the fire pit is preferable to living.



**Calculating utilities over time**

In the fire pit world, the utilities of the exit states don't change as the game goes on. They are +1 and -1 for the entire game and the living reward is also the same throughout the game. However, what if time were a factor in the game: the agent needs to reach the end by some number of iterations or the game ends. That changes the game, and possibly the optimal policy for the agent.

**Finite vs. infinite horizon**

**Finite horizon:** There is a fixed time N after which nothing matters, e.g. the game ends. If we were looking at additive rewards, where Uh is the utility of a particular state sequence, then Uh([s0, s1, s2 ... sn+k]) = Uh([s0, s1, s2 ... sn]), k > 0. A policy where the agent goes the long way around the maze to avoid the fire pit may not work in a finite horizon problem.

**Infinite horizon:** There is no time limit on the problem and the optimal action depends only on the current state, not the time when that state is encountered. These are the problems we're going to address. The challenge then is setting up the problem so that the state sequence is not infinite. If rewards are positive and additive, there's no incentive to finish the search. However, if rewards are discounted, then they are worth less over time and the agent needs to balance the risk of going near the fire pit, for example, with the decreasing reward for the longer it takes to reach the goal.

**Discounted Rewards:**

In search problems with discounted rewards, there is an additional parameter, γ, where 0 < γ < 1, that is applied to positive rewards on each step in the search. The utility of a state sequence is Uh = R(s0) + γR(s1) + γ2R(s2) ... Rewards in the current time are worth more than future rewards, and rewards in the distant future are insignificant.

If we look back at the fire pit problem, and instead of having R(s) < 0, let R(s) > 0. In an infinite horizon problem with no discounting, the search would never converge to a solution because there would be no incentive for the agent to ever exit the world. The policy would never reach a terminal state and the MDP would fail. Discounted rewards make it easier to use standard approaches to solving MDPs.

(This equation doesn't include the probability of the state transitions yet, but our final policy calculation will.)

**Calculating optimal policies**

We compare policies by comparing the expected utilities when executing the policy. The utility of a state is defined in terms of the state sequences that might follow it, and state sequences depend on policy π.

Let *St* be the agent's state at time *t* in policy π. Then, the expected utility obtained by executing policy π in state *s* is the sum of all future discounted rewards from the current state. (*S0* is the state the agent is in at the current time.)

Uπ(s)=E[∑t=0∞γtR(St)]

This equation can be a little confusing because it's easy to interpret as t = 0 as the initial state of the search. However, it's actually the agent's current state.

The optimal policy is the one with the highest expected utility

π∗s=argmaxaUπ(s)

which we can also write as:

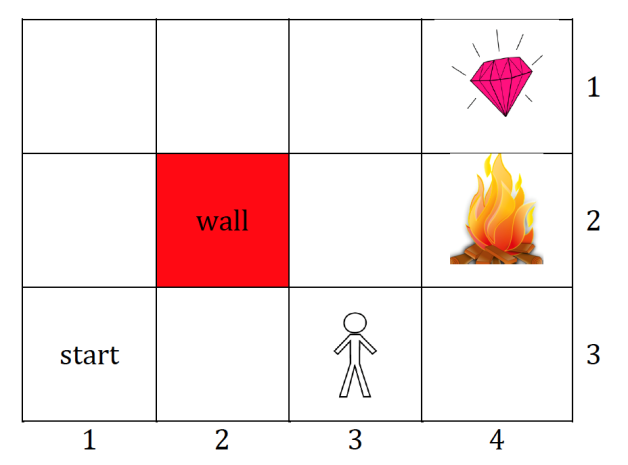
π∗s=argmaxa∑s′P(s′|s,a)U(s′)

**Bellman equation for utilities**

The utility for a state is the reward for the current state plus the sum of the discounted utility for the next state:

U(s)=R(s)+γ argmaxa∑s′P(s′|s,a)U(s′)

**Example: Bellman equation for state (3, 1)**



U(3,1)=R(3,1)+γ max{.8U(2,1)+.1U(3,2)+.1U(3,1),.9U(3,1)+.1U(2,1),.9U(3,1)+.1U(3,2),.8U(3,2)+.1U(2,1)+.1U(3,1)}

**Value Iteration**

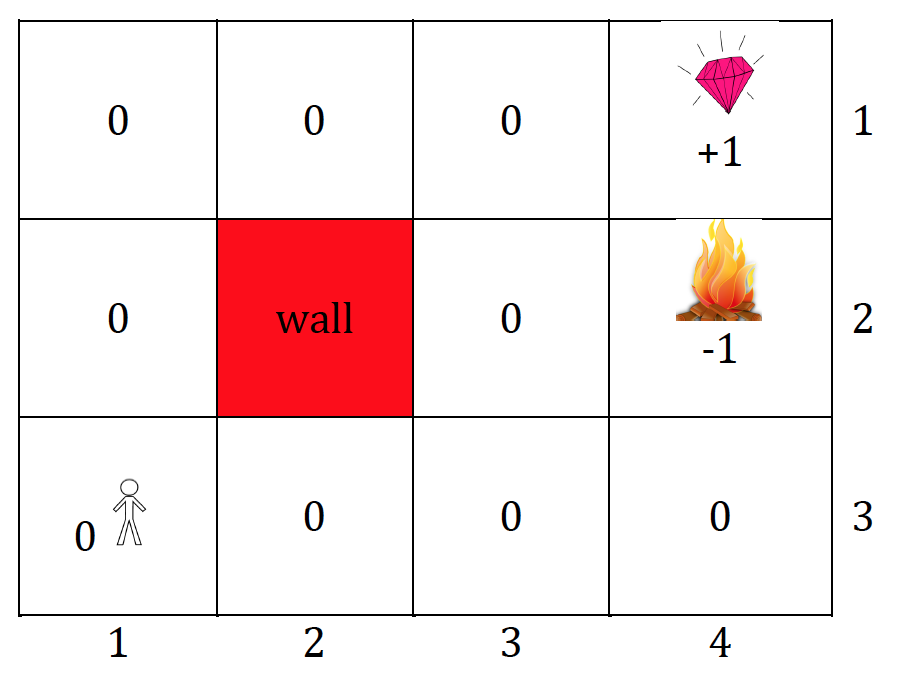
The value iteration algorithm is used to assign utilities to each state using the Bellman equations. Each state has its own equation. The algorithm begins by assigning arbitrary initial values for utilities to each state, and then iteratively updates those utilities until an equilibrium is reached.

Initialize U(s) = 0, for all s.

Let Ui(s) be the utility value for state *s* at *ith* iteration of algorithm.

Ui+1=R(s)+γ max∑s′T(s,a,s′)Ui(s′)

If we start with initial utility of 0 for all states, the utility for the +1 reward state and -1 reward state will be the reward values after one iteration of value iteration.



 Let the living reward R = 0, and γ = .9, 80% success, 10% of moves to left and 10% to right.

On U2(s), U(1,3) will be updated to be > 0.

U2(1,3)=R(s)+γ max[(.8∗1.0+.1∗0+.1∗0)right,(.8∗0+.1∗0+.1∗1.0)up,(.8∗0+.1∗0+.1∗1.0)down,(.8∗0+.1∗0+.1∗0)left,]