* **Problem-solving agent** - when the correct action to take isn’t obvious and agent needs to plan ahead and consider a sequence of actions that form path to goal state
  + The computational process it undertakes is called **search**
  + Use **atomic** representations (state of world is considered as whole, with no internal structure visible
  + **Planning agents** use factored or structured representation of states

If the environment is **unknown** the agent can’t do better than executing an action at random.

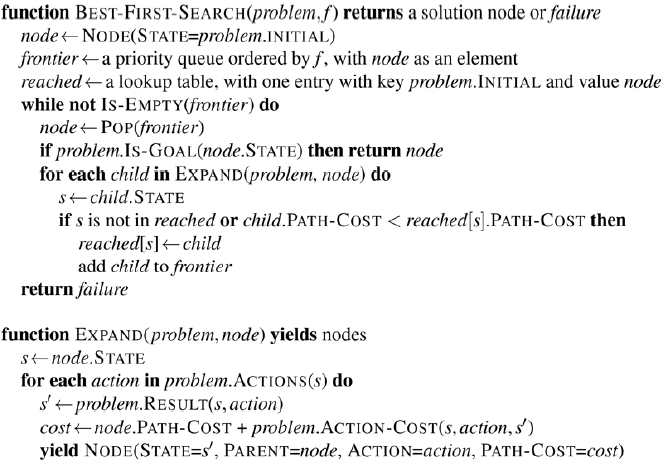
* Search environments are
  + Episodic
  + Single-agent
  + Fully-observable
  + Deterministic
  + Static
  + Discrete
  + Known
* Four-phase problem solving process for a **known** environment
  + **Goal Formulation**: goals organize behavior by limiting objectives and hence the actions to be considered
  + **Problem Formulation:** agent devises description of the states & actions necessary to reach the goal
  + **Search:** agent simulates sequences of actions in its model until it finds a sequence of actions that reaches the goal
  + **Execution**: Agent executes actions in the solution

In a **fully observable**, **deterministic**, **known environment**, the solution to any problem is a fixed sequence of actions.

* **Open-loop system** - if the model is correct, once the agent has found a solution it can ignore its percept while executing the actions because the solution is guaranteed to lead to the goal.
  + If there is a change that the model is incorrect or the environment is nondeterministic, the agent should consider a **closed-loop** approach that monitors percepts.
* Formal definition **of search problem**
  + **State space**: set of possible states the environment can be in
  + **Initial state**: state the agent starts in
  + **Goal state:** set of one or more states. Can also be defined by a property that applies to many (potentially infinite!) states
  + **Actions**: Actions available to an agent
    - Given State returns finite set of actions that can be executed in s
      * Each of the actions are **applicable** in S.
    - **Absolute** actions vs. **egocentric** actions (defined relative to the viewpoint of the agent)
  + **Transition model**: describes what each action does
    - returns the state that results from doing action *a* in state *s*
  + **Action cost function**: Gives numeric cost of applying action *a* in state *s* to reach state *s’*
    - Problem-solving agents should use cost functions that reflect their own **performance measures**
  + **Path**: sequence of actions
  + **Solution**: path from the initial state to the goal state
    - Assume action costs are additive
  + **Optimal solution**: lowest path cost among all solutions
    - Assume all action costs will be positive
  + **Graph**: state space can be representation in which the vertices are states and directed edges between them are actions
* **Abstraction** - the process of removing detail from a representation
  + **Model**: an abstract mathematical description
  + Certain considerations are left out of models because they’re irrelevant
  + Needs the right level of detail
  + **Valid**: can we elaborate any abstract solution into a solution in the more detailed world?
  + **Useful**: is carrying out each action in the solution easier than the original problem?
* **Standard vs. Real-world**
  + **Standardized problems** are intended to illustrate/exercise problem solving methods
    - Grid World
    - Sokoban Puzzle
    - Sliding-tile Puzzle
    - Knuth’s square-root, floor, factorial integer problem
  + **Real-world problems** are ones whose solutions people use
    - Route-finding
    - Touring: set of locations must be visited
    - VLSI Layout
    - Robot Navigation
    - Automatic assembly sequencing
* **Search algorithm** - takes search problem as input and returns a solution or indication of failure
  + Some algorithms superimpose **search trees** over the state-space graph, forming various paths from the (**root**) **initial state** trying to find a path that reaches the goal state
    - Each **node** in the search tree corresponds to a **state** in the state space and **edges** in the search tree correspond to **actions**
* **State Space vs. Search Tree**
  + **State space**
    - Describes possibly infinite set of states in the world
    - Describes actions that allow transitions from one state to another
  + **Search tree**
    - Describes paths between the states reaching toward the goal
    - May have multiple paths to any given state
    - Each node in the tree has unique path back to the root
* **Search Tree Operations**
  + **Expanding nodes**: considering all available actions for a state, using to see where the actions lead, and generating a new child/successor node for each resulting state
  + **Frontier**: set of unexpanded nodes.
    - Separates 2 regions of the state-space graph, interior region where every state is expanded, and exterior region of states that haven’t been reached
  + **Reaching**: Any state that has had a node generated has been reached **whether or not that node has been expanded**.

The essence of search is following up one option now and putting others aside for later**. How do we decide which node from the frontier to expand next?**

* **Best-First Search -** choose a node *n*, with minimum value of some **evaluation function**
  + Each iteration
    - Choose a node on the frontier with minimum *f(n)*
    - Return state if it’s a goal state
    - Otherwise to generate child nodes
    - Each child node is added to the frontier if not reached
      * Re-added if it’s not being reached with a path that has a lower path cost than any previous path



* **Search data structures** - search algorithms require data structure to keep track of search tree
  + **Node**
    - *node.*STATE - state to which node corresponds
    - *node.*PARENT - node in the tree that generated this node
    - *node.*ACTION - action that was applied to the parent’s state to generate the node
    - *node.*PATH-COST - the total cost of the path from the initial state to this node. Synonymous with *g(node)*
  + **Queues:** data structure to store the frontier
    - **Priority queue**: Pops node with minimum cost according to some evaluation function *f*. Used in best-first search
    - **FIFO queue:** Pops the node that was added to the queue. Used for BFS
    - **LIFO queue (stack)**: Pops the most recently added node. Used for DFS
  + **Reached states** can be stored as a lookup table where each key is a state and each value is the node for that state
* **Redundant paths - repeated states** in the search tree, can be generated by a **cycle (loopy path)**
  + Even if the state space is finite, the complete search tree is infinite because there’s no limit to how often one can traverse a loop
  + **Cycle:** special case of redundant path
  + **Algorithms that cannot remember the past are doomed to repeat it**
    - How can we approach this issue?
      * **Remember all previously reached states**
        + Used in best-first search
        + Allows us to detect all redundant paths
        + Keep only the best path to each state
        + Appropriate for state spaces with many redundant paths
        + Preferred choice when the table of reached states will fit in memory
      * **Don’t worry about repeating the past**
        + Some problem formulations where it’s rare/impossible for two paths to reach same state

Can save memory space if we don’t track reached states and don’t check for redundant paths

* + - * + **Graph search**: Search algorithms that check for redundant paths

Best-First Search is graph search

* + - * + **Tree-like search**: doesn’t check for redundant paths
      * **Check for cycles, but not for redundant paths in general**
        + Each node has a chain of parent pointers

Check for cycles by following up chain of parents

See if the state at the end of the path has appeared earlier in the path

No need for additional memory

Some implementations follow the chain all the way up

Eliminates all cycles

Other implementations only follow a few links

Takes constant time

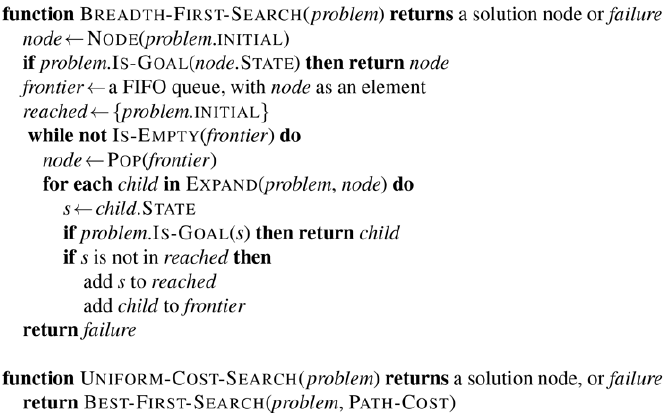
Eliminates short cycles

Relies on other mechanisms to deal with long cycles

* **Measuring problem-solving performance**
  + **Completeness**: is the algorithm guaranteed to find a solution when there is one and to correctly report failure when there isn’t?
    - **A complete algorithm must be capable of systematically exploring every state that is reachable from the initial state**
    - Finite state spaces
      * As long as we keep track of paths that are cycles, eventually reach every reachable state
    - Infinite state spaces
      * Algorithm needs to make sure it can eventually reach any state that is connected to the initial state.
      * In an infinite state space with no solution a sound algorithm *needs to keep searching forever because it can’t know if the next state will be a goal*
  + **Cost Optimality (Admissibility)**: does the algorithm find a solution with the lowest path cost of all solutions?
  + **Time Complexity**: how long does the algorithm take to find a solution? Can be measured in seconds or abstractly by the number of states/actions considered
  + **Space Complexity**: how much memory is needed by the algorithm to perform the search?
    - Measured by
      * *|V|*  is the number of vertices (state nodes) of the graph
      * |*E*| is the number of edges (distinct state/action pairs)
      * Appropriate when graph is an explicit data structure
        + E.g. Map of Romania
    - Measuring complexity in an implicit state space
      * *d* - **depth** or number of actions in optimal solution
      * *m -* max number of actions in any path
      * *b* - **branching factor** or number of successors of a node that need to be considered

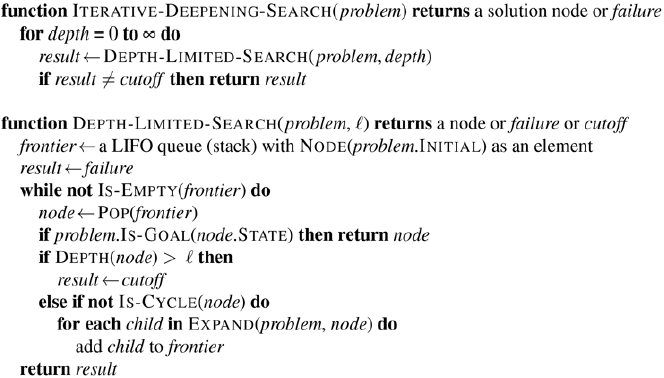
**An uninformed search algorithm is given no clue about how close a state is to the goal**

* **Breadth-first search (BFS)** - appropriate when all actions have the same cost. The root node is expanded first, then the successors of the root, then their successors (left to right)
  + Systematic strategy, so it’s complete even on infinite spaces
  + Can implement BFS as a call to best-first-search where *f(n)* is the depth of the node (the # of actions it takes to reach the node)
  + **Additional efficiency**
    - FIFO queue will be faster than priority queue
      * Always gives us correct order of nodes
    - Reached can be a *set of states* rather than mapping from states to nodes
      * Once we’ve reached a state we can’t ever find a better path to teh state
    - **Early goal test**
      * Check whether a node is a solution as soon as it’s generated
      * **Late goal test**: waiting till node is popped of the queue to check if the node is a solution
  + BFS always finds solution with minimal number of actions because when it’s generating nodes at depth *d* it’s already generated all the nodes at depth *d - 1*.
    - **Cost optimal for problems where all actions have the same cost but not for problems that don’t**
    - **Complete in either case**
    - Space/Time Complexity
      * Imagine searching a uniform tree where every state has *b* successors
      * Suppose solution is at depth *d*
    - Memory requirements are a bigger problem for BFS than the execution time

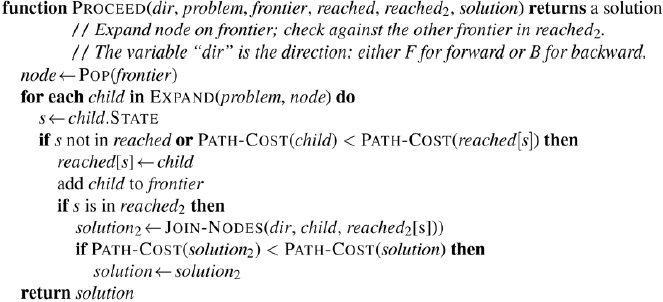
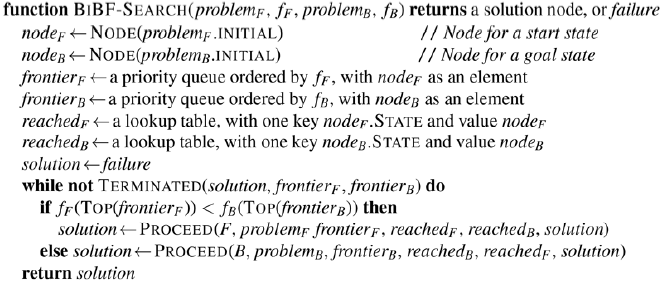


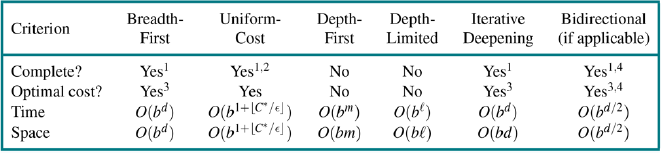
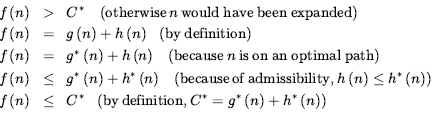
**Exponential-complexitysearch problem can’t be solved by uninformed search for any but the smallest instances**

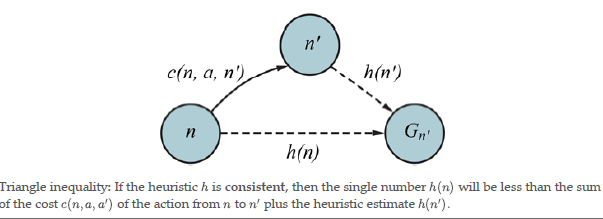
* **Dijkstra’s Algorithm or Uniform Cost Search (UCS) -** appropriate when actions have different costs
  + Evaluation function is the cost of the path from the root to the current node
  + Algorithm only tests for goals when it expands a node, not when it generates a node
    - Because if another path has a lower cost, it’ll be a better solution
  + **Complete**
  + **Cost-optimal**
    - First solution it finds will have a cost that is *at least* as slow as the cost of any other node in the frontier
  + Considers all paths systematically in order of increasing cost
    - Never gets caught in a single infinite path assuming action costs are > 0
  + Complexity
    - *-* cost of the optimal solution
    - - lower bound on the cost of each action ()
      * Can be much greater than (space/time complexity of BFS)
        + UCS can explore large trees of actions with low costs before exploring paths involving a high-cost and useful action
    - If all action costs are equal is just
* **Depth-first search (DFS)** - always expands the deepest node in the frontier
  + Evaluation function is the negative of the depth
  + Usually implemented as a tree-like search instead of graph search
    - Doesn’t keep table of reached states
  + **Not cost-optimal**
    - Returns the first solution it finds, even if not cheapest
  + **Incomplete**
    - For finite trees
      * Efficient and complete
    - For acyclic state spaces
      * Inefficient, but systematic
    - Cyclic state spaces
      * Can get stuck in infinite loop
        + Some implementations check each new node for cycles
    - For infinite state spaces
      * Not systematic
      * **Incomplete**
  + **Best for problems where a tree-like search is feasible**
    - Much smaller needs for memory
    - Frontier is very small
    - No *reached* table
  + **Complexity**
    - For finite trees
      * Memory complexity of
        + *b -* branching factor
        + *m* - maximum depth of tree
  + **Backtracking search -** variant of DFS that uses even less memory
    - Only one successor is generated at a time rather than all successors
    - Each partially expanded node remembers which successor to generate next
    - Successors are generated by modifying the current state description directly
      * As opposed to allocating memory for brand new state
      * Reduces memory requirements to just one state description and a path of actions
    - Option of maintaining efficient set data structure for the states of the current path
      * Allows us to check for cyclic path in instead of
    - Must be able to *undo* each action when we backtrack
* **Depth-limited search** - keeps DFS from wandering down infinite path
  + Version of DFS where a depth limit is supplied
    - Treat all nodes at depth *l* as if they have no successors
  + **Complexity**
    - Time:
    - Space:
  + Can’t keep it from wasting time on redundant paths in general
    - If we look up a few links in the parent chain, can catch most smaller cycles
    - Longer cycles are handled by *l*
  + Good depth limit can be chosen based on knowledge of the problem
    - For most problems, we won’t know a good depth limit until the problem is solved
* **Iterative deepening search** - solves the problem of picking a good value for *l* by trying all values until a solution is found or depth-limited search returns failure value
  + **Complexity**
    - Space: when there is a solution or for finite state spaces with no solution
    - Time: when there is a solution or when there is none
  + **Optimality**
    - for problems where all actions have the same cost
  + **Complete** on finite acyclic state spaces
    - Complete on any finite state space when we check nodes for cycles all the way up the path
  + Iterative deepening doesn’t store nodes in memory, it just repeats the previous levels
    - For many state spaces most nodes are in the bottom level, so repeating is OK
    - Total number of nodes generated in the worst case
  + **Hybrid approach**
    - Runs BFS until almost all available memory is consumed then runs iterative deepening from all nodes in frontier
  + Iterative deepening is the preferred uninformed search method when the search state space is larger than can fit in memory and the depth of the solution is not known



* **Bidirectional search -** simultaneously searches forward form the initial state and backwards from the goal state hoping that the 2 searches meet
  + Motivation: is much less than
  + Need to keep track of two frontiers, two tables of reached states
  + Need to reason backwards
    - If *s’* is a successor of *s* in the forward direction, need to know that *s* is a successor of *s’* in the backward direction
    - Solution when the two frontiers collide
  + **Bidirectional best-first search**
    - Node to be expanded next is always the one with a minimum value of the evaluation function across either frontier
    - When the evaluation function is path cost, we get bidirectional UCS
    - If the cost of the optimal path is no node with cost > /2 will be expanded



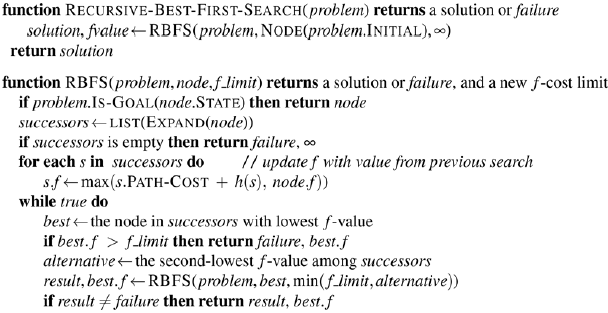
* **Comparing uninformed search algorithms** for tree like search versions which don’t check for repeated states
  + For graphs that do check, the main differences
    - DFS is complete for finite state spaces
    - Space time complexities are bound by the size of the state spaces
      * Number of vertices and edges 
* **Informed (Heuristic) Search Strategies -** strategies that use **heuristic functions** *h(n)* to approximate the location of goals
  + *h(n)* - estimated cost of the cheapest path from the state at node *n* to a goal state
* **Greedy best-first search -** form of best-first search that expands the first node with the lowest *h(n)*
  + E.g. for distance-based problems *h(n)* is the **straight-line distance** (SLD) heuristic.
    - Takes a certain amount of world knowledge to know SLD is a useful heuristic
  + Algorithm is called “**greedy**” because it doesn’t always find the optimal solution
  + **Complete** in finite spaces **incomplete** in infinite spaces
  + **Space/Time Complexity** is
    - With a good heuristic certain problems can reach
* **A\* search** - best-first search that uses evaluation function
  + *g(n)* - path cost from the initial state to node *n*
  + *h(n)* - estimated cost of the shortest path from *n* to goal state
  + **A\* is complete**
    - Assuming all action costs > 0 and state space either has a solution or is finite
  + **Properties of heuristic**
    - An **admissible heuristic** is one that *never overestimates* the cost to reach a goal (aka the heuristic is **optimistic**)
    - Admissible heuristic = A\* is cost-optimal
    - Proof by contradiction
      * - optimal path cost
      * But algorithm returns path cost
      * Then there must be some node *n* which is on the optimal path and is unexpanded
      * - the cost of the optimal path from the start to *n*
      * - the cost of the optimal path from *n*  to the nearest goal
      * First and last lines form a contradiction
        + Supposition that the algorithm could return a suboptimal path must be wrong
        + **It must be that A\* returns only cost-optimal paths**
    - **Consistency:** *h(n)* is consistent if for every node *n* and every successor of *n’* of *n* generated by action *a*



* + - * If the heuristic is consistent:
        + It’s admissible (so A\* is cost optimal)
        + The first time we reach a state it will be on an optimal path
        + Never have to re-add a state to the frontier
        + Never have to change na entry in reached
      * If the heuristic is inconsistent
        + May end up with multiple paths reaching the same state
        + If each new path has a lower path cost than previous one, will end up with multiple nodes for that state in the frontier
      * To combat downfalls of inconsistency
        + Implementations of A\* take care to only enter a state into frontier once
        + If better path is found all successors of the state is updated

Nodes need child and parent pointers

* + - * + Better to avoid inconsistent heuristics?
      * If heuristic is inadmissible A\* may or may not be cost optimal
  + A\* expands all nodes that can be reached from the initial state on a path where every node on the path has *f(n) < C\**. These are **surely expanded nodes**
  + A\* might then expand some nodes on the right of the “goal contour” before selecting a goal node
  + A\* expands no nodes with *f(n) > C\**
  + **A\* with a consistent heuristic is optimally efficient** in the sense that any algorithm that extends search paths from the initial state and uses the same heuristic information must expand all nodes that are surely expanded by A\*
  + A\* is efficient because it prunes away search tree nodes that are not necessary for optimal solution
* **Search contours** - visualizing a search by drawing contours in the state space
  + A\* adds nodes in concentric bands of increasing *f*- cost (bands will stretch toward goal state, narrowly focused)
  + UCS has contours of *g-*cost (circular around start state no preference in direction)
  + As you expend a path, *g* costs are **monotonic - the path cost always increases as you along a path**
    - Action costs are positive
    - Always get concentric contour lines that don’t cross each other
  + Not obvious if *f(n)* will always monotonically increase
    - Only if the heuristic is consistent
    - Monotonic heuristic = consistent heuristic
* **Satisficing search** - we can explore fewer nodes if we’re willing to accept suboptimal solutions
  + Allowing A\* to use **an inadmissible heuristic** (one that overestimates) risks suboptimal solutions but the heuristic can be more accurate and expand less nodes
  + **Weighted A\*:** where we weight the heuristic value more heavily
    - Solution might be costlier, but search time is faster
  + **Bounded suboptimal search**: look for a solution that is guaranteed to be within a constant factor *W* of the optimal cost
  + **Bounded-cost search** look for a solution whose cost is less than some constant *C*
  + **Unbounded-cost search** accept solution of any cost as long as it’s found quickly
    - Speedy search: version of greedy best-first search that uses a heuristic the estimated number of actions required to reach a goal regardless of the cost of those actions
* **Memory-bound search -** A\*’s main issue is the use of memory which is split between frontier and reached state
  + Instead of storing a state as a node in the frontier and in the table of reached state, keep the state in only one of the two spaces
    - Complicates the algorithm
  + Remove states from reached when we can prove they’re not needed
  + Keep reference counts of the number of times a state has been reached and remove it from reached table when there are no more ways to reach the state
  + **Beam search**: limits the size of the frontier
    - Keep only the *k* nodes with the best *f* scores and discard other nodes
    - Makes the search incomplete and suboptimal
    - Alternatively keep every node with *f* scores within some delta of the best *f* score
  + **Iterative-deepening A\* (IDA\*)**
    - Benefits of \* without the requirement to keep al reached states in memory
    - Cutoff is the *f* cost instead of the depth
      * The cutoff value is the smallest *f* cost of any node that exceeded the cutoff on previous iteration
  + **Recursive best-first search**
    - Resembles recursive DFS, but uses *f\_limit* to keep track of *f* value of the best alternative path available from any ancestor of the current node
      * If current node exceeds limit, recursions unwinds back to alternative path
    - Replaces *f* value of each node along the path with backed-up value (best *f* value of children)
    - Remembers *f* value of the best leaf in the forgotten subtree
    - Somewhat more efficient than IDA\* but suffers from excessive node regeneration
    - Optimal if the heuristic is admissible
    - Would perform better if algorithm was implemented to use all available memory
      * Memory-bound A\* (MA\*) or Simplified MA\* (SMA\*)
        + Proceeds like A\*, expanding best leaf until memory is full
        + Always drops the worst leaf node (highest *f* value)
        + Backs up value of forgotten node to its parent
        + Regenerates subtree only when other paths have been shown to look worse than the path that’s been forgotten
        + Complete if there is any reachable solution
        + Optimal if any optimal solution is reachable



* **Bidirectional heuristic search**
  + Unidirectional best-first search using *f(n) = g(n) + h(n)* gives A\* search that is guaranteed to find optimal cost solutions (with admissible *h*)
  + No guarantee for bidirectional best-first search using the same *f(n)*
  + Any proof of efficiency will have to consider pairs of does
  + Consider a path from initial state to a node *m* and a backward path from the goal to node *m*
    - Cost of such a path must be *at least* as large as the same of the path costs of the two parts
    - Cost must be *at least* as much as the estimated *f* cost of either part
  + Never know for sure which node is best to expand, so no bidirectional search algorithm can be guaranteed to be optimally efficient
  + **Front-to-end search**: Where the heuristic estimates the distance to the goal and estimates the distance to the start
  + **Front-to-front** attempts to estimate the distance to the other frontier
  + If we have a a good heuristic adding bidirectional search doesn’t help much
  + With an average heuristic bidirectional search expands fewer nodes and is preferred
* **Heuristic Functions**
  + **Effective branching factor b\***
    - Gauges quality of a heuristic
    - If total number of nodes generated by A\* for a problem is *N* and the solution depth is *d* then b\* is the branching factor that uniform tree of depth would have to have in order to contain *N + 1*  nodes
    - Experimental measures of b\* on small set of problems can provide a good guide of the heuristic’s overall usefulness
    - Well-designed heuristic would have a b\* value close to 1
  + **Effective depth**
    - Characterizes the effect of A\* pruning
    - Gives accurate predictions for total search cost
  + If one heuristic is better than the other, we say that heuristic **dominates** the other
    - A\* using a more dominating heuristic will never expand more nodes than A\* using a less dominating heuristic
* **Generating heuristic from relaxed problems**
  + A problem with fewer restrictions on the actions is called a **relaxed** problem
  + State space graph of the relaxed problem is a supergraph of the original state space because the removal of restrictions creates added edges in the graph
  + Any optimal solution in the original problem is also a solution in the relaxed problem
  + The relaxed problem may have better solutions if the added edges provide shortcuts
  + The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
  + Because the derived heuristic is an exact cost for the relaxed problem, it must be consistent
  + If a problem definition is written down in formal language it is possible to construct relaxed problems automatically
    - Crucial that the relaxed problems generated by this technique can be solved essentially without search
  + If a collection of admissible heuristics is available
    - provides the best heuristic
    - Because is admissible and consistent*, h* is admissible and consistent and *h* dominates all component heuristics
      * Only downfall is that *h(n)* takes longer to compute
* **Generating heuristics from subproblems: pattern databases**
  + Admissible heuristics can be derived from the solution cost of a subproblem
  + **Pattern databases**: store exact solution costs for every possible subproblem instance
    - Compute admissible heuristic *h* for each state encountered during a search by looking up the corresponding subproblem in the database
    - Database is constructed by searching back from the goal and recording the cost of each new pattern encountered
    - Expense of this search is amortized over subsequent problem instances
      * Makes sense if we expat to be asked to solve many problems
* **Generating heuristics with landmarks**
  + **Precomputation** of optimal path costs
    - Only needs to be done once and can be amortized over search requests
    - Could generate perfect heuristic by precomputing and storing the cost of the optimal path between every pair of vertices
  + Better approach is to choose a few landmark points from the vertices
    - For each landmark *L* and for each other vertex *v* in the graph, compute and store the exact cost of the optimal path from *v* to *L*.
    - Given the stored *C\** tables, create efficient (but inadmissible) heuristic
      * Minimum over all landmarks of the cost of getting from the current node to the landmark, and then to the goal
    - If optimal path goes through landmark, the heuristic is exact, if not inadmissible (overestimates cost to goal)
    - In A\* search with exact heuristics once a node on the optimal path is reached, every node expanded from then on is on the optimal path
  + **Shortcuts**: artificial edges in the graph that define an optimal multi-action path
    - Differential heuristic (one that subtracts)
      * Landmark that is somewhere out beyond the goal
  + Get better results if we pick landmark points that are spread out and not too close to each other
* **Learning to search better**
  + **Metalevel state space:** captures the internal state of a program that is searching in an object-level state space
    - Each action in the metalevel state space is a computation step that alters the internal state
  + **Metalevel learning** can learn from missteps and avoid unpromising trees
* **Learning heuristics from experience**