- **Depth-First Search**: Completing the search of one path before exploring the other. Treats the frontier as a stack. If we use list and *select* is made to select the first element of the list, *add_to_frontier* will add the neightbors in front of the list.

```
add_to_frontier(NF, F1, F2):- append(NF, F1, F2).
```

- **Breadth-First Search**: Always takes the path with fewest arcs to expand. Treats the frontier as a queue. If we use list and *select* is made to select the first element of the list, *add_to_frontier* will add the neightbors at the end of the list.

```
add_to_frontier(NF, F1, F2):- append(F1, NF, F2).
```

- **Lowest-Cost-First Search**: Need to have a function $c(n)$ that returns the cost of reaching a node n. This strategy requires the expansion of the lowest cost path first. Treats the frontier as a priority queue. If we use list and *select* is made to select the element with the lowest cost first, it does not matter how *add_to_frontier* is implementd.

Space and Complexity of the Blind Search Strategies: Important factors in deciding which strategy to use.

Depth-First	Breadth-First
Might not find the solution	Guarantee to find a solution if one exists if branching factor is finite
Linear in size of the path being explored	Exponential time and space in size of the path being explored
Search is unconstrained until solution is found	Search is unconstrained by the goal

Lowest-cost strategy: similar to breath-first.

4 Heuristic Search – Informed Search Strategies

Idea: Taking into account the goal information and (if available) knowledge about the goal. At any iteration, the "most promising" node – one, that probably leads to the goal – is selected to expand the frontier. Represent as a *heuristic function*, h, from the set of nodes into non-negative real numbers, i.e., for each node n, $h(n) \geq 0$.

$h(n)$ is *underestimate* if it is less than or equal the actual cost of the lowest- cost path from n to the goal.

Example: For the robot delivery, the straight-line distance between the node and the goal is a good heuristic function, which is underestimate.

Example: For the SLD search graph, the number of atoms in the query is a heuristic function.

- **Best-First Search**: Always select the element that appears to be the closest to the goal, i.e., lowest $h(n)$. Frontier is treated as priority queue. The simplest way to implement it: 'select' chooses the first element of the list and 'add_to_frontier' sorts the new list by the heuristic function.

```
select(N,[N|F1],F1).

add_to_frontier(NN, F1, F2):- append(NN, F1, F), sort_by_h(F, F2).
```

- **Heuristic Depth-First Search**: Like depth-first, but use $h(n)$ in deciding what branch of the search tree to explore. Implementation:'select' chooses the first element of the list and 'add_to_frontier' sorts the list of neighbors before putting them in front of the frontier list.

```
select(N,[N|F1],F1).

add_to_frontier(NN, F1, F2):- sort_by_h(NN, F), append(F, F1, F2).
```

- **A* Search**: Selecting the next node based on the actual cost and the estimate cost. If the actual cost to the node n is $g(n)$ and the estimate cost from n to the goal is $h(n)$, the value $f(n) = g(n) + h(n)$ will be used in selecting the node to expand the frontier. This method is implemented by a priority queue based on $f(n)$.

```
select(N,[N|F1],F1).

add_to_frontier(NN, F1, F2):- append(NN, F1, F), sort_by_h(F, F2).
```

Best-First	Depth-First	A*
Might not find the solution	Might not find the solution	Guarantee to find an optimal solution if one exists and branching factor is finite
Exponential time and space in size of the path being explored	Linear in size of the path being explored	Exponential time and space in size of the path being explored

5 Refirements to Search Strategies

Idea: Deals with cycles in the graph

- **Cycle checking**: Before inserting new paths into the frontier, check for their occurrence in the path. If the path selected to expand is $\langle n_0, \ldots, n_k \rangle$ and m is a neighbor if n_k we add to the frontier the path $\langle n_0, \ldots, n_k, m \rangle$ if m does not occur in $\langle n_0, \ldots, n_k \rangle$.
  - easy to implement in depth-first (one extra bit); set when visits; reset when backtracks;
  - need more time in exponential space strategies
- **Multi-path Prunnning**: Before inserting new paths into the frontier, check for the occurrence of new neighbors n the frontier. (The following implementation only considers the nodes – rather than the paths; modifications are needed for considering pahs.)

```
select(N,[N|F1],F1).

add_to_frontier(NN, F1, F2):- diff(NN, F1, F), append(F, F1, F2).
```

  Need to be done carefully if shortest/lowest cost path need to be found. In A*, *monotone restriction* is sufficient to guarantee that the shortest path to a node is the first path found to the node.

  monotone restriction: $|h(n') - h(n) \leq d(n', n)$ where $d(n', n)$ is the actual cost from n' to n.

  Subsumes cycle checking. Preferred in strategies where the visited nodes are explicitely stored (breath-first); not prefered in depth-first searchs since the requirement of space required.
- **Iterative deepening**: Instead of storing the frontier, recompute it. Use depth-first to explore paths of 1, 2, 3, ... arcs until solution is found. When the search fails *unnaturally*, i.e., the depth bound is reached; in that case, restart with the new depth bound.

- Linear space in size of the path being explored.
- Little overhead in recomputing the frontier.

Iterative deepening A*: Instead of using the number of arcs as the bound, use $f(n)$. Initially, $f(s)$ is used (s is the start node with minimal h-value). When the search fails unnaturally, the next bound is the minimal f-value that exceeded the previous bound.

- **Direction of Search**: forward (from start to goal), backward (from goal to start), bidirectional (both directions until meet). The main problem in bidirectional search is to ensure that the frontiers will meet (e.g. breath-first in one direction and depth-first in the other).
  - **Island-driven Search**: Limit the places where backward and forward search will meet (designated *islands* on the graph). Allows a decomposition of the problem in group of smaller problems. To find a path between s and g, identify the set of islands $i_0, \ldots, i_k$ and then find the path from s to i_0, from i_j to i_{j+1}, and finally from i_k to g.
  - **Searching in a Hierarchy of Abstractiosn**: Find solution at different level of abstraction. Details are added to the solution in refinement steps.
- **Dynamic Programming**: construct the perfect heuristic function that allows depth-first heuristic to find a solution without backtracking. The heuristic function represents the *exact costs* of a minimal path from each node to the goal. This will allow us to specify *which arcs to take* in each step, which is called a **policy**. Define

$$dist(n) = \begin{cases} 0 & if\ \ is\_goal(n) \\ min_{\langle n,m\rangle \in A}(|\langle n,m\rangle| + dist(m)) & otherwise \end{cases}$$

  where $dist(n)$ is the actual cost to the goal from n.

  $dist(n)$ can be computed backward from the goal to each node. It can then be stored and used in selecting the next node to visit.

  - $dist(n)$ depends on the goal;
  - $dist(n)$ can be pre-computed only when the graph is finite;
  - when $dist(n)$ is available, only linear space/time is needed to reach the goal;

6 Constraint Satisfaction Problem (CSP)

A CSP is given by a set of variables, a domain for each variable, and a set of constraints. The goal is to find an assignment for the variables that either (i) satisfies a set of constraints; or (ii) has the least cost associated with it.

A problem of the first type is called *satisfiability probllem*. It is referred to as an *optimization problem* if it belongs to the second type.

There are two types of constraints:

- Hard constraints: those that must be satisfied by the variable assignment;
- Soft constraints: those that are preferred.

CSP can be viewed as a graph searching as well. Two ways:

- A graph with the nodes are possible variable assigments. Neighbors of a node are those that can be obtained from it by changing the value of one variable. Alternately,

- First, we order the variables as $\langle V_1, \ldots, V_n \rangle$ and the graph is defined by nodes of the form $\langle v_1, \ldots, v_k \rangle$ where $k = 1, \ldots, n$, v_i belongs to the domain of the variable V_i, and represents the assignment for v_i at the node. Neighbors of a node $\langle v_1, \ldots, v_k \rangle$ is the nodes $\langle v_1, \ldots, v_k, v_{k+1} \rangle$ where v_{k+1} is the assignment for V_{k+1}.

Definition 6.1 *A CSP is given by*

- *A set of variables $V_1, \ldots, V_n$;*
- *For each variable V_i, a domain D_i (possible values for V_i);*
- *For satisfiability problem: a set of constraints on various subsets of the variables;*
- *For optimization problem: a cost function that gives a cost to each assignment.*

Definition 6.2 *A solution to a satisfiability problem is a variable assignment that satisfies all constraints.*

A solution to an optimization problem is a variable assignment that minimizes the cost.

Example 6.1 (Knap-sack problem) A bank-robber is lucky to get into a room with Gold, Silver, and Bronze coints each of which worths US$ 650, US$ 200, and US$50, respectively. He want to make the most out of the opportunity. He has a knap-sack that can contain several coints but he can only carry at most 48 pounds. Each Gold, Silver, and Bronze coint weights 10, 3, and 4 pounds, respectively. What should he do?

Question: How do we represent this as a CSP problem?

The robber has to select three numbers: G - the number of gold coints; S - the number of silver coints; B - the number of bronze coints. *So, there are three variables!*

He cannot take more than 4 gold coints. Also, it is impossible to take just half of a coint since he does not have time. Thus, G can take the values 0, 1, 2, 3, or 4. That is, the domain of G, that we denote by D_G is $\{0, 1, 2, 3, 4\}$.

Similarly, $D_B = \{0, 1, 2, \ldots, 16\}$ and $D_G = \{0, 1, 2, \ldots, 12\}$.

Since the guy cannot carry more than 48 pounds, we have the constraint:

$$10G + 3S + 4B \leq 48$$

The guy wants to maximize the value of his catch, so he wants:

$$650G + 200S + 50B \Rightarrow \mathbf{max}$$

Without the maximization of the value of the robber's catch, we have a satisfiability problem. With it, we have an optimization problem with the cost function that assigns the value $-(650G + 200S + 50B)$ to a variable assignment

Example 6.2 (Scheduling) Let A, B, C, D, E denote the time the activities a, b, c, d, e are scheduled, respectively. They should be at the time points $1, 2, 3,$ or4. So, the domains are

$$D_A = D_B = D_C = D_D = D_E = \{1, 2, 3, 4\}.$$

The constraints:

$$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge (C < D) \wedge (A = D) \wedge$$
$$(E < A) \wedge (E < B) \wedge (E < C) \wedge (E < D) \wedge (B \neq D).$$

Classification of CSP:

- *Finite domains CSP:* The domains of the variables in the problem are finite – each domain consits of a finite number of discrete values.
- *Binary CSP:* Each constraint involves only one or two variables.

The knap-sack problem in the previous example is a FCSP but not a binary CSP. The scheduling problem is a finite and binary CSP.

A satisfiability FCSP can be represented by a Datalog program without rules – where in every clause terms are constants and in a query, ever term is a variable.

The complexity of CSP is NP-hard. Different approaches to solving a CSP:

- Find an algorithm that works well on typical cases even though the worst case might be exponential;
- Find special cases for which there are efficient alogrithms exist;
- Find efficient approximation algorithm;
- Develop parallel and distributed algorithm.

6.1 General algorithms soving CSP

Generate-and-Test Algorithms: For a problem with variables $V_1, \ldots, V_n$ and the corresponding domains $D_1, \ldots, D_n$, we can generate all possible variable assigments and test each of them one-by-one for satisfiability/minimality.

Use: the Cartesian product $D = D_1 \times \ldots \times D_n$ is the search space for the solution. The search space grows very fast!

The knap-sack problem has $5 \times 17 \times 13 = 1105$ possible variable assigments.

The scheduling problem has $4 \times 4 \times 4 \times 4 \times 4 = 1024$ possible variable assigments.

Backtracking Algorithms: Use partial assignment of variables, check for satisfiability of constraints as soon as it is possible. Eliminate huge search space if a constraint is not satisfied. This improves over generate-and-test algorithms. Corresponds to depth-first search.

Example: If we add one more constraint "*The robber wants at least one gold coint*" to the example, then the partial assigment $G = 0$ could not be a part of any solution which means that we can eliminate $17 \times 13 = 221$ possible solution assigments from the search space.

Consistency Algorithms:

Constraint network: a graph (N, A); each node $X \in N$ corresponds to a variable and its domain is attached to the node; an arc between two nodes X and Y, $P(X, Y)$, represents the constraint imposed on X and Y.

A node is *domain consistent* if the no value in the domain is ruled impossible.

$\langle X, Y \rangle$ is arc consistent if for each value of X in D_X there is some value for Y in D_Y such that $P(X, Y)$ is satisfied. A network is arc consistent if all its arcs are arc consistent.

AC-3: an algorithm to solve CSPs using arc-consistency algorithm. The key idea: makes the network arc consistent. (Detail in Page 153). The steps:

- Create a queue of arc constraints;
- Try to resolve the inconsistent arc; Update the queue every time a domain of a variable is changed.
- Stop when cannot continue or the queue is empty.

Hill Climbing: Address the problem of the huge size of the search space. Require a heuristic function that gives a value to a total variable assignment. Start out at one assignment and iteratively improves the value yield by the heuristic function until no improvement can be made. The implementation of `hill_clim` is as follows.

```
hill_clim(N, N):-
    neightbors(N, NN), best(N, NN, N).

hill_clim(N, S):-
    neightbors(N, NN), best(N, NN, M),
    h(M)>h(N), hill_clim(M,S).

best(N, [], N).
best(N, [M|R],B):-
    h(N)>=h(M), best(N, R, B).
best(N, [M|R],B):-
    h(N)<h(M), best(M, R, B).
```

Here, $h(N)$ is the heuristic value of node N.

Characteristics of hill climbing:

- Might not get to global maximum.
- Often used in situations with multiple dimentions where systematic search would take too long.
- For continuous domains, use gradient-descent when trying to find minimum value; use gradient-ascent when trying to find maximal value. The formula for next value of X_i is

  $$v_i - \eta \frac{\partial h}{\partial X_i} \text{ (min)} \quad \texttt{or} \quad v_i + \eta \frac{\partial h}{\partial X_i} \text{ (max)}$$

  Need to pay attention to the value of η.

Randomized Algorithms: Try to avoid the pitfall of hill climbing (finding only local maximum/minimum). Two possibilities:

- *random-restart hill climbing*: pick a value, start hill climbing to find the local maximal; then pick another value, find local maximal; etc.. Disadvantage: lots of time spent on hill climbing.
- *two-phase search*: pick a number of values, start hill climbing from the maximal value; Disadvantage: still could not find the global maximum.

Other methods:

- **Simulated Annealing**: Combine hill climbing and random selection. At any iteration of the search, select a neighbor randomly and use a heuristic function in deciding whether or not to visit that neighbor.
- **Beam Search**: Like hill climbing but maintain k best nodes. The nodes do not interact.
- **Stochastic Beam Search**: Maintain k nodes as beam search but the nodes are selected randomly (nodes with higher heuristic value is more likely to be chosen).
- **Genetic Algorithm**: Similar to stochastic beam search but introduce a new way to create new elemments of the next iteration. The new operation, called *cross-over*, produces new element from a pair of nodes.