Heuristic Search & Optimization

Diego Vicente Martín

November 11, 2017

Contents

1 Dynamic Programming

- **Dynamic programming** is a bottom-up procedure that suggests to store sub-results (memoization) so that they can be reused later on to compute the optimal solution of a given problem.
- The general procedure is:
  1. Break the global problem into **similar sub-problems** and charazterize their structure in a simple way.
  2. **Recursively** define the value of an optimal sub-problem solutions.

3. Compute the values of optimal subproblems in a **bottom-up** fashion.
4. **Construct** an optimal solution from the computed intermediate results.

2 Linear Programming

- A linear programming problem is in **canonical form** if:

  1. The objective function is in form of **maximization**.
  2. All the constraints are **inequalities of the form** $\leq$.
  3. All the decision variables are **non-negative**.

  The algebraic representation of the canonical form would be:

  $$\max Z = c^{\mathsf{T}} x$$

  $$Ax \leq b, x \geq 0$$

  Where $A_{m \times n}$ is the **matrix of technological coefficients** of m inequalities and n decision variables; $x_{n \times 1}$ is the column vector of **decision variables**; $b_{m \times 1}$ is the vector of **resources** and $c_{n \times 1}$ is the vector of **costs or benefits**.

- A linear programming problem is in **standard form** if:

  1. The objective function is in form of **maximization or minimization**.
  2. All the constraints are **equations**.
  3. All the decision variables are **non-negative**.
  4. The vector of **constant resources** does **not contain negative** components.

  The algebraic representation of the standard form would be:

  $$\max / \min Z = c^{\mathsf{T}} x$$

  $$Ax = b, x \geq 0$$

- We must take into account some **transformations** that can be useful when solving linear programming problems:
  - A minimization problem can be converted to a maximization one changing the sign of the objective function:

$$\min Z = \sum_{j=1}^{n} c_j x_j \longrightarrow \max Z' = -\sum_{j=1}^{n} c_j x_j$$

  - We can change the direction of an inequality multiplying both sides by -1:

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \longrightarrow -\sum_{j=1}^{n} a_{ij} x_j \leq -b_i$$

- Any inequality can be converted into an equality by adding or resting a nonnegative varibale (**slack variable**) with a null coefficient (zero) in the objective function.

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \triangleq \sum_{j=1}^{n} a_{ij} x_j + s_i = b_i$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \triangleq \sum_{j=1}^{n} a_{ij} x_j - s_i = b_i$$

2.1 Graphical Resolution

- Procedure:
  1. Represent the problem in **canonical form** (though it is not strictly necessary).
  2. Draw a **cartesian coordinate system** where each axis represents a decision variable.
  3. Represent each **constraint as a region** (that may be unbounded).
  4. The intersection of all regions is the **feasible region**, the solution space F.
  5. **Evaluate the objective function** in the extreme points and pick up the one that maximizes the objective function.

- If the problem does not have a unique optimal solution, then it is said to have **alternative optimal solutions**. This is easily detected using the graphical method, if the curve of **isoprofit** / **isocost** is parallel or identical to one of the constraints whose extreme points are optimal solutions.
- A problem is found to be unfeasible if and only if the **region of feasible solution is empty**: $F = \emptyset$.
- This method is only useful with a maximum of 3 decision variables.

2.2 Simplex Method

- Procedure:
  1. Start the problem by converting the task to **standard form** and adding the necessary **artificial variables** to the problem (with ∞ coefficient in Z), in order to ensure an initial solution.
  2. Compute the **basic variables**:
     - $B_i = \{x_a, x_b, x_c \dots\}$ with n vectors (to be a square matrix)
     - $x_{B_i} = B_i^{-1} b$
  3. Select the entering variable by using the **entering rule**:
     - $y_n = B_i^{-1} a_n$
     - $z_n - c_n = c_{B_i}^{\mathsf{T}} y_n - c_n$
     - Entering variable is n where $\min\{z_n - c_n\} \forall n \in B$
  4. Select the leaving variable by using the **leaving rule**:
     - $\theta = min\{\frac{x_{B_i}}{y_n}\}$ where n is the entering variable column of the basis **as long as** $\theta > 0$.
  5. The solution is given by the value of $x^* = x_{B_f}$ once the iterations are over, and the value of the objective function $z^* = z_f$
- **Interpretation of the results**:
  - If a **slack variable is part of** x_f, there is an excess amount (if the variable's constraint was a lower bound) or a deficit (if it was an upper bound) in the resources.
  - One thing that points out that a problem is unfeasible is the fact that an **artificial variable is part of** x_f.

2.3 Duality

- A linear programming problem is in **symmetric form (of maximization)** if:
  - The objective function is in form of **maximization**.
  - All the constraints are **inequalities in the form of** $\leq$.
  - All the decision variables are **non-negative**.
  - In case it is a minimzation problems, all constraints must be in $\geq$ form.
- If a **primal problem** is

$$\max Z = c^{\mathsf{T}} x$$

$$Ax \leq b, x \geq 0$$

  then its **dual problem** is:

$$\min W = b^{\mathsf{T}} x'$$

$$A^{\mathsf{T}} x' \geq c, x' \geq 0$$

  If the problem is in symmetric form, and has an optimal solution to a basis B, then $x'T = c_B^{\mathsf{T}} B^{-1}$ is an optimal solution to the dual problem.
- The **economic interpretation** states: the dual variable x_i' indicates the contribution per unit of the i^{th} resource b_i to the variation in the current optimal value z of the objective.

2.4 Modelling

- There are some well-known problems for which there are some already created patterns.
- The **demand problem**, where a_i is the capacity of the i origin, b_j is the demand of the j destination, and c_{ij} is the unitary cost of deliver from i to j; the problem can be modelled like:

- Objective function: $\min Z = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$
- The capacity is not exceeded: $\sum_{j=1}^{m} x_{ij} \leq a_i$
- The offer is satisfied: $\sum_{i=1}^{n} x_{ij} \geq b_i$
- The variables are positive integers: $x_{ij} \in \mathbb{Z}^+$

- The **assignment problem**, where n people must be assigned to m tasks to be performed in an optimal way provided the unitary cost of each assignment by each person, c_{ij}. This problem uses the variables as boolean variables, where $x_{ij} \leftrightharpoons 1, 0$ depending if a person i has been or not assigned to a certain task j.
  - Objective function: $\min Z = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$
  - One task is assigned to each person: $\sum_{j=1}^{m} x_{ij} = 1, \ \forall i$
  - One person is assigned to each task: $\sum_{i=1}^{n} x_{ij} = 1, \ \forall j$

- The **network flow problem**, where we have a graph $G = (V, E)$ and a function of cost, and we must find the cheapest path from one vertex x_1 to another x_n. In this problem we will also use the variables as boolean variables, where $x_{ij} \leftrightharpoons 1, 0$ depending if we have taken the edge that connects i and j.
  - Objective function: $\min Z = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$
  - Only one edge leaves the origin: $\sum_{j=1}^{n} x_{1j} = 1$
  - Only one edge arrives at the goal: $\sum_{j=1}^{n} x_{in} = 1$
  - There are as many entering edges as leaving ones in every node: $\sum_{j=1}^{n} x_{il} - \sum_{j=1}^{n} x_{lj} = 0, \ \forall l \in V$

3 Logical Satisfiability

- A propositional formula is in **Conjunctive Normal Form (CNF)** if it's in form of:

$$\bigwedge_{i=1}^{N} \bigvee_{j=1}^{N} \ell_{ij}$$

- A literal ℓ consists of an atom (one variable) in its assertion (x) or negation ($\bar{x}$).
- A literal ℓ is **pure** in F if $\bar{\ell}$ does not appear in any clause of F.
- A **tautology** is an always true propositional formula for any instantiation of its atomic expressions.

3.1 Resolution Method

- **Davis-Putnam Algorithm**:
  1. Select a literal $\ell \in F$
  2. Apply $Res\{F, \ell\}$ and write down the variable used and clauses involved.

$$Res\{(p \vee r), (\bar{p} \vee r)\} = (r \vee s)$$

$$Res\{p, \bar{p}\} = \{\emptyset\}$$

  3. If it results in an empty clause $\{\emptyset\}$, then F **is not satisfiable**.
  4. If it results in $F = \emptyset$, end the problem. If not go back to step 1.
  5. Consider the list of variables and clauses in inverse order, calculating whether $\top$ or $\bot$ for each of them.

- **Davis-Putnam-Logemann-Loveland Algorithm**: consists of applying reduction on a formular F in form of a tree. The reduction function consists in taking a variable and exploring the possibilities of having the variable asserted or negated while the whole formula is still true.

4 Constraint Processing

- A **constraint network** $R = (X, D, C)$ consists of a finite set of values (X) defined over domains (D) that contain the possible value for each variable and a set of constraints (C).
- We can say that x_i is **arc consistent** with x_j if and only if for each value a_i there exists a value a_j such that $(a_i, a_j) \in R_{ij}$. If there is a value that does not participate in a relation, it can be eliminated. Arc consistency is not a bidirectional relationship.
- A constraint R_{ij} is **path consistent** relative to variable x_k if and only if for every pair $(a_i, a_j) \in R_{ij}$ there is a value $a_k \in D_k$ such that $(a_i, a_k) \in R_{ik}$ and $(a_k, a_j) \in R_{kj}$. Path consistency is indeed a bidirectional relationship.

5 Search

- A **problem space** is composed by:
  - The **states set**, that contains every possible state of every search problem.
  - The **operators set**, that contains every single action that can be performed to the problem.
  - The **initial state**.
  - The **goal**.
- There are two important factor:
  - The branching factor, b, that is a property of the state graph.
  - The depth, d, that is a property of the problem.
- We can perform several algorithms:
  - **Breadth First Search (BFS)**: searches through all nodes of a certain depth before exploring the ones deeper. It's behavior is best understood if we think of adding nodes to a queue to be expanded. It is complete, optimal (with equal costs among operators), efficient if goals are close to the root, but it consumes exponential memory.

  - **Depth First Search (DFS)**: expands nodes depth-wise, and thus it needs backtracking technique if the solution is not found and even a depth limitation if the branch being expanded is infinite. By itself, it is not complete, it is not optimal, but it is efficient as long as there is no cycles if the goals are away from the root.
  - **Iterative DFS** performs a DFS with a depth limitation that is incremented each if the solution is not found in the current nodes. If we set the depth increment to 1, it basically becomes a BFS. It is complete, if the increment is one it is optimal, but it can generate a lot of duplicated nodes.
  - **Depth First Branch-and-Bound** performs a DFS until it finds a solution or the cost of all paths exceed the cost of the best solution found so far. Every time a solution is found, a new bound is set at its depth.

- If we have no information, we perform a blind search, but we can also have a perfect information or a **heuristic**, that is a partial knowledge about the problem/domain that permits solving the problem efficiency in this domain.

- We can find heuristics by solving simplifiend models of the problem (**constraint satisfaction**), adding constraints, using a probabilistic estimation or reasoning by analogy or metaphor.

- A heuristic is **admissible** if never overestimates the real cost of reaching a goal.

- Some **informed search algorithms** are:

  - **Hill Climbing**: chooses the next node to expand based on which one of them has the best heuristic value. It is a greedy method, so it can find some problems like local maxima or minimas, plateaus, ridges... So it is not complete. It is useful and efficient with consistent heuristic functions and we can solve its problems using backtracking, or random restarts.
  - **Beam Search**: it works as Hill Climbing, although it expands k nodes each iteration. Opening the window improves the possibilities of escaping from plateaus and finding shorter paths. Usually the higher value of k, the better the solutions found, although it is not always like that.

- **Best First Search (BFS)**: always expands the most promising node according to a given rule at a given time and orders the nodes by its rule in a sorted queue to be expanded.
- **A*** is a BFS algorithm what uses the function $f(n) = g(n) + h(n)$ to evaluate the nodes, where $g(n)$ is the cost of the travelled arcs up to node n and $h(n)$ is the value of the heuristic for that node. It is complete, admissible (as long as the succesors fot the nodes are finite and the heuristic is admissible too) and a more informed heuristic will expand less nodes than a bad one.
- **IDA*** is a BFS search that uses a A* search evaluation in an iterative deepening way, using a value $\eta = \min\{f(i)\}$ as an upper bound to search for a solution. It is complete, it is admissible, although it can be time-costly (exponential) although its memory complexity is linear.