CHAPTER 6

Approximation

In the last chapter when we discussed the complexity (Section 5.1.1) we found problems that where too expensive to find a general or exact solution. Please recall that the *traveling salesman* problem is of exponential time complexity that makes it unpractical to solve for more than 15 cities. Reality suggests a pragmatic approach and accepts approximate solutions as not every city is directly connected with every other city. Sometimes it is possible to solve a simplified version of the problem only, e.g. taking into account only interstate highways.

For some other problems we don't care about the exact solution. An approximate value will suffice in many cases. Suppose we want to know the average age of all people in the USA. We really do not care about the exact value, a rounded number will do.

In case that not all information is available to get the correct answer then a limited answer is better than nothing. Looking for the cheapest bargain flight is constraint to the availability of offers in the Web.

These examples show that a pragmatic approach is based on reality constraints, experiences, or a kind of tradeoff. The search for a flexible process to obtain a solution with a certain tradeoff will be presented in the present chapter.

CHAPTER OBJECTIVES

- To know Polya's heuristic problem solving techniques.
- To identify some application domains for heuristics.
- To know how to use histograms.

6.1 Heuristic Computation

The word **heuristic** originates from the Greek ευρωκειν (heuriskein = to find, to discover) and means to find a solution.

Heuristics are the study and application of heuristic methods. **Heuristic methods** are iterative problem-solving techniques in which the most appropriate solution is selected from several but not all possible solutions. The solution may be refined at successive stages during the solution process.

The Hungarian mathematician George Polya **?** made this idea popular in his book “How to solve it”. For solution finding he recommends that his students do the following:

- If you are having difficulty understanding a problem, try drawing a picture.
- If you can’t find a solution, try assuming that you have a solution and seeing what you can derive from that ("working backward").
- If the problem is abstract, try examining a concrete example.
- Try solving a more general problem first (the "inventor’s paradox": the more ambitious plan may have more chances of success).

The idea is so general that his propositions have been adopted in many disciplines. In psychology a set of very simple, but efficient *rules of thumb* have been used as heuristic to explain how people make for instance buy decisions.

In the case of laws heuristic methods are used where a case-based judgement would be impractical. For that reason general ages for legal drinking, voting, or driving a car have been established by law. The intention is that people should be mature enough to be able to do these things in a responsible manner. But for certain individuals the legal age comes to late or to early. Indeed, the “legal age” is just a rule of thumb.

In computer science the fundamental goal is finding the provable best algorithm with respect to run time and solution quality. A heuristic in this context means finding an algorithm that gives up these goals but still provides acceptable results. An heuristic therefore is a kind of strategy to find a good solution with acceptable effort.

For example, using heuristics will usually find pretty good solutions, but there is no proof that this will hold in every situation; or the algorithm usually runs reasonably quickly, but without guaranty that this will always be the case. In deed, one can find exotic problems where an heuristic will produce bad results or run too slowly. However, these instances might never occur in practise or are of little relevance. Therefore, the use of heuristics is very common in real world software applications.

Let’s develop a simple heuristic for the visualization of a function. The goal is to provide an adequate plot for any one-dimensional function in a bounded interval.

It is clear that we have to distinguish between bounded and unbounded functions. The latter can not be plotted with any linear scale on a sheet of paper. Under the precondition that the function values are bounded, we can scale the y-axis to fit on our paper or display. If the function values grow very fast, a logarithmic scale could help. If the function is periodic like the trigonometric function one period will suffice.

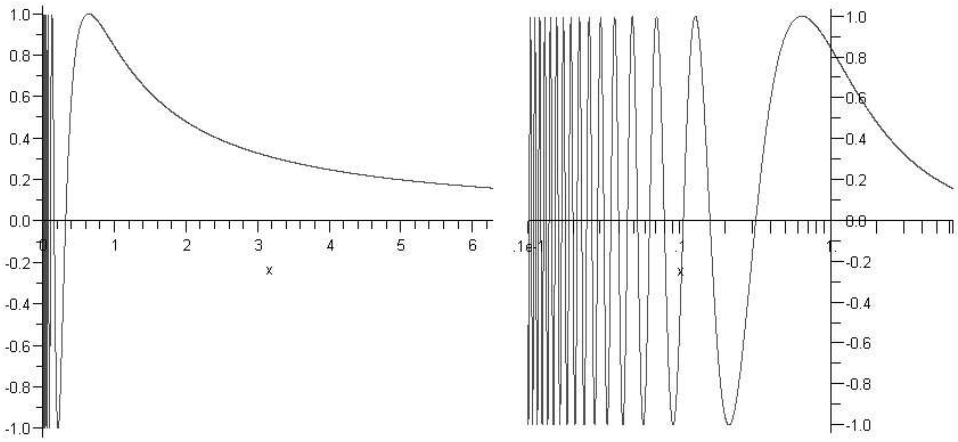

Figure 6.1 Plots of $sin(1/x)$, without and with heuristic scaling

When the function values in a subinterval of the definition domain are changing rapidly we should consider either the interesting subinterval only or chose a nonlinear scale to show the interesting region in detail.

In Figure 6.1 the function $sin(1/x)$ is plotted. The plot on the left side cannot show what happens for small values of x, but on the right side graph an heuristic is applied using a semi-logarithmic scale. It clearly exhibits that the function oscillates faster as x approaches 0. Please note that even with a logarithmic x-scale the function oscillates faster towards 0 than the stretching of the scale.

Another issue is the performance of function visualization. To provide a most accurate plot we need to compute as much function values as the display media resolution. It is more efficient to compute less values and interpolate the values in between. If we do so with equidistant x-values the result will not be optimal if the function makes a sharp curve. Therefore a good heuristic is to use more values when the curve is narrow and less values when the curve is wide. This is shown in Figure 6.2 where the function $sin(x)$ is computed for 57 values. Most of the values are computed around the extremes at $x = \pi/2$ and $x = 3\pi/2$ where the function has the narrowest curves. Please note that according to our previous heuristics we plot only one period.

Problems in *robotics* typically require heuristics for fast decision making or complex computer vision analysis. In both cases speed is dominant over accuracy. But there are many more application domains for heuristics that have been identified in Computer Science:

- Algorithm development
- Shortest path-problems
- Computational performance
- Artificial Intelligence
- Expert Systems

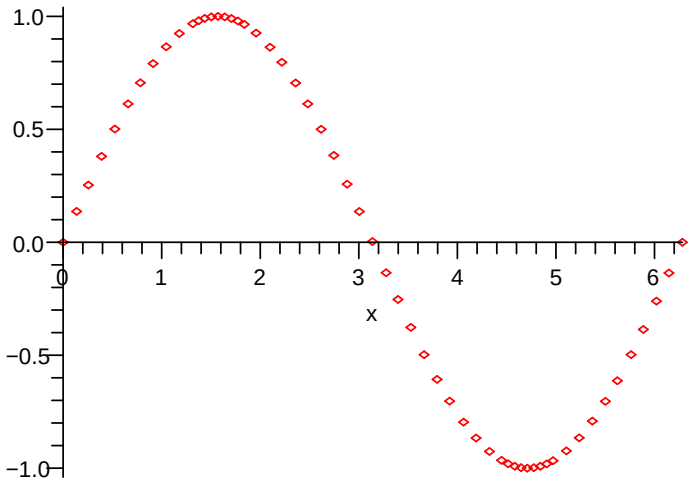

Figure 6.2 Plots of $sin(x)$ with heuristic computation of function values

- Virus detection

A general algorithm with provably good run-time and a provably optimal solution is hard to find in these application domains. In *algorithm development* heuristics are used to find algorithms that solve subtypes of a problem in a non-general but efficient way.

Heuristics for *shortest path-problems* mean to find cost functions to order the path sequence in a way that the search algorithm is more efficient.

Heuristics for *shortest path-problems* mean to find an ordering of the possible pathes in a way that the shorter pathes are likely to appear first. This makes the search algorithm more efficient because the evaluation of the pathes can be stopped when a sufficiently good path is found.

More general than the shortest path-search are problems with a high branching factor b and a potential search-depth d. The "cost" for finding the goal node is in general b^d. Heuristics tell that in many situations the branching factor could be reduced and the search-depth might be limited so that *computational performance* can be considerably enhanced. We will investigate in Subsection 6.3.1 the *MiniMax* decision algorithm with branching factor 2 and possible search depth d.

In *Artificial Intelligence* researchers tried to use heuristic methods to build programs that can improve its behavior and judge its results. Professor Douglas Lenat (MIT) and his colleagues designed EURISKO to explore new areas of knowledge. It is guided by heuristics that suggest plausible actions to follow or implausible ones to avoid. Other heuristics look for reoccurring patterns in results, propose new heuristics, and rate the value of both new and old heuristics. In this way EURISKO evolves better behaviors, better internal models, and better rules for selecting among the models.

Expert Systems are specialized Artificial Intelligence applications where rules for the analysis of information is gathered and "feed" into the program in order to become a knowledge inference system with better abilities than a single expert. Expert systems have proved successful in medical diagnosis, technical trouble-shooting and other limited domains.

The term heuristic is used in connection with *virus detection* if an unknown pattern or behavior is suspected as a virus because it is similar to an already known virus.

6.2 Dynamic Programming

Dynamic programming tries to solve problems more efficient that have a recursive solution. Recursive algorithms use the *divide and conquer* principle in a top-down manner. With this approach intermediate results are often recomputed which makes the algorithm inefficient. The dynamic programming uses a bottom-up approach to build the solution by solving subproblems only once and combining them recursively. This approach assumes that

- the problem can be broken down into reusable subproblems
- the optimal solution can be constructed from the optimal solution of subproblems (Bellman optimum principle).

Under this preconditions the process consists of three steps:

- Break the problem into smaller subproblems
- Solve the subproblems *recursively* from solutions of smaller subproblem.
- Build up an optimal solution for the original problem by combining the solutions for the subproblems

The algorithm is recursive because the solution of a subproblem is found by solving a smaller subproblem within the same process step.

As we will see in the following example during the solution process results for subproblems should be remembered because they are needed several times to find the global solution.

As example for dynamic programming we use the route planning problem. Say we want to drive from New York (NY) to Los Angeles (LA). Looking for the optimal route means checking all routes from NY to LA. In many cases we have to check the same road sections for multiple routes. To solve subproblem s_3 we can make use of the solution for subproblem s_2 and s_1. And solution of subproblem s_3 can be used to solve problem s_4. So if we recall the already checked roads we do not need to compute them again. Combining the best sections we will eventually find the optimal route. Figure 6.3 illustrates the problem with a simplified situation.

Please observe, that the global solution is an optimization problem. It minimizes the distance (or time depending on the score function) to travel from NY to LA.

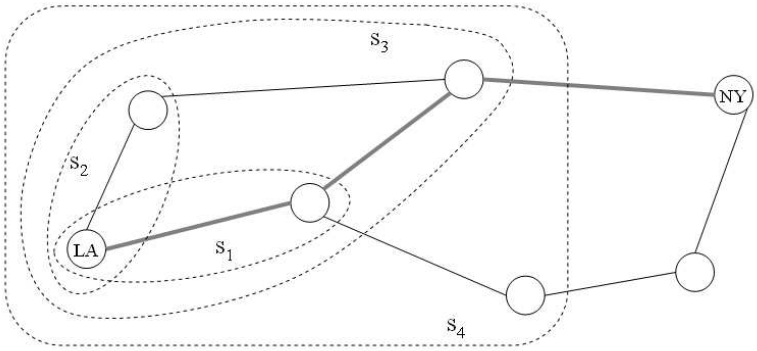

Figure 6.3 Finding the best route (shortest path)

6.3 Decision Theory

An other kind of "optimization problem" is choosing the optimal *game strategy*. Special classes of two player games have been extensively studied and strategies have been developed to choose the best "move" in a certain situation toward the goal of the game. On each player's turn a new decision has to be taken. The strategy is to anticipate all possible moves and chose the best in the current situation.

A game is called *zero-sum* if the *gain* or "advantage" of one player is the *loss* or "disadvantage" of the other player. Examples for zero-sum games are:

- Chess
- Reversi
- Tic-tac-toe
- Poker

It is easy to see, that the gain of one player is the loss of the other. But, there are multi-player games like the German "Mensch ärgere dich nicht" which is similar to "Sorry!" that are zero-sum, too.

6.3.1 MiniMax Algorithm

For the zero-sum two-player games the *MiniMax* strategy has been developed to *Mini*mize the *Max*imum loss. Each possible game position is evaluated and a value v associated with it. Favorable situations for player one get an high value and favorable situations for the second player get a low value. The strategy is to maximize the gain for the player in turn. So the goal function alternates between minimizing and maximizing the value with each turn.

The *minimax* algorithm has four steps:

1. build decision tree from current position
2. evaluate leaves

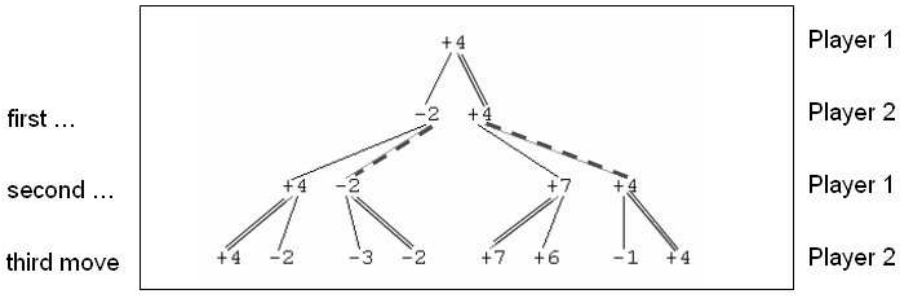

Figure 6.4 Example MiniMax decision tree of search depth three

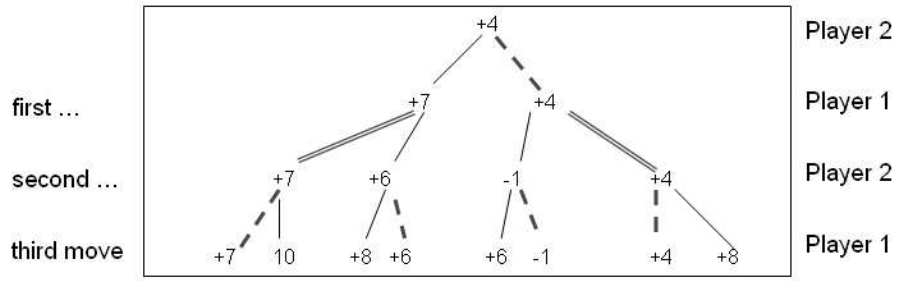

Figure 6.5 MiniMax decision tree for the next move

3. copy evaluation recursively towards current position (root) taking:
   - the maximum leaf value if it is first player's turn
   - the minimum leaf value if it is second player's turn
4. choose next position as:
   - node with maximum value if it is first player's turn
   - node with minimum value if it is second player's turn

An example game play is visualized in figure 6.4 as decision tree. The nodes represent game situations and the edges are the possible moves. Nodes on the odd levels correspond to the first player's turn and nodes on the even levels to the second player's turn. The red edges lead to a max-value, and the blue edges lead to a min-value.

When we analyze figure 6.4 we see that the double line to the node with the maximum value 4 will be chosen. On the next decision the minimum value (dotted line) will be chosen. Figure 6.5 illustrates this move. Please note that again the search depth is 3 which means that we look 3 moves ahead.

The minimax algorithm has to be repeated until the game is finished. The cost is growing exponentially with the depth of the decision tree, but a deep

tree is important to find the "best" decision. This observation is the motivation for refining the minimax algorithm.

6.3.2 Alpha-beta Pruning

The minimax algorithm evaluates and looks at many nodes which do not influence the root node. If we compare a certain node with other nodes and find out that this node is not as good as the other, then there is no need to look at this *branch* (subtree) further because it does not influence the outcome.

The idea can be illustrated having two pockets with valuables. We will get the item of least value from the pocket we choose. If we can look into the pockets before making the choice and find a knife in the right pocket and an apple in the left. Then, there is no need to look into the left pocket again until we find anything less valuable in the first pocket than an apple.

The *alpha-beta pruning* updates two values *alpha* and *beta* which indicate the worst case result for player A and B. As we only consider zero-sum games for two players the worst case for B is also the best case for A.

Here is the alpha-beta pruning algorithm in pseudo-code:

```
eval (node, α, β)
   if node is a leaf
      return value of node
   end if
   if node is a minimizing node
      for each child of node
         β = min (β, eval (child, α, β)) ! recursion
         if β <= α
            return α
      end for loop
      return β
   end if
   if node is a maximizing node
      for each child of node
         α = max (α, eval (child, α, β)) ! recursion
         if β <= α
            return β
      end for loop
      return α
   end if
end eval
```

In the example[1] of figure 6.6 only 12 of 18 nodes are evaluated because the values of the cutout nodes do not improve the result.

The algorithm starts at the root with the maximum alpha-beta window $(-\infty, +\infty)$. The window is handed over to all child nodes at the beginning. The algorithm starts evaluating the leaf nodes 1, 2, 3 by the max-function. The best value 10 is handed to the parent node and propagated as beta value to its neighbors. This value adapts the other node's beta value having children

[1] taken from http://de.wikipedia.org/wiki/Alpha-Beta-Suche

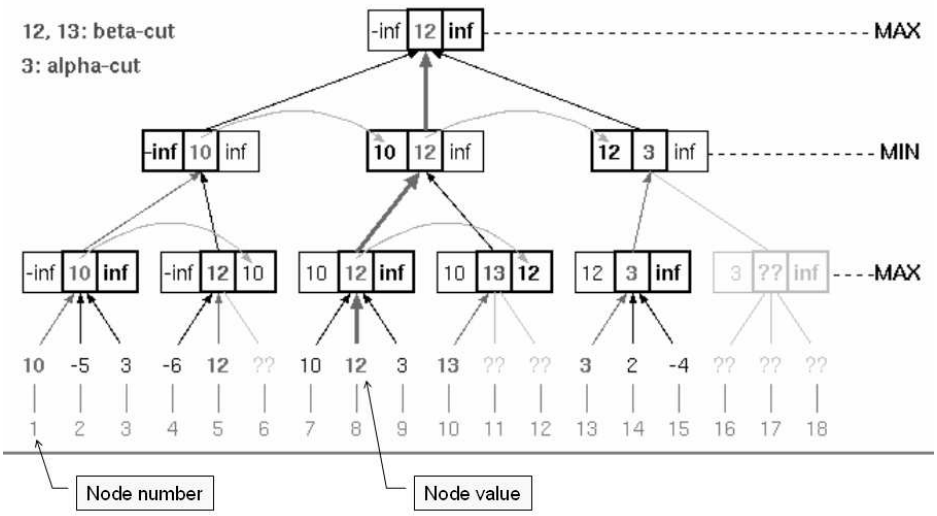

Figure 6.6 Alpha-beta pruning example

4, 5, 6. Node no 5 evaluates to 12 that exceeds the beta limit. Thus, the next leaf node no 6 needs not to be evaluated because the opponent (using a minimizing strategy) prefers to choose node 1 which is less than leaf node 5. The situation with the subtree of minimizing node [12,3,inf] is analogous. It is not necessary to evaluate its leaf nodes further because the maximizing player will never choose this branch because he found a better subtree (the one under node [10,12,inf]) where at least the result 12 is guaranteed.

In this algorithm it is important to narrow the alpha-beta window $|\alpha - \beta|$ as soon as possible to reduce the number of evaluations. In other words, try to cut branches early. Therefore it is important to evaluate potential "best" moves first. We use *heuristics* for finding candidate nodes that are likely to make an alpha or beta cutoff.

An other optimization is a variable search depth. Depending on the cost and the "promise" of a node the search depth will be adjusted. This *iterative deepening* is often used in conjunction with the *aspiration window*. In this variant the alpha-beta window is initially chosen with the same values as the parent node. If this window appears to be too small, the search is repeated with a maximum window.

6.4 Shortest Path Algorithms

Path algorithms are used for route planning, traveling salesman problem, traffic management, and other related tasks. These are very important problems for any logistic task. If you want to drive from New York (NY) to Los Angeles (LA) looking for a[2] shortest route like in Figure 6.3 the criterion to minimize is the distance from NY to LA.

[2]in fact there may exist more than one shortest route

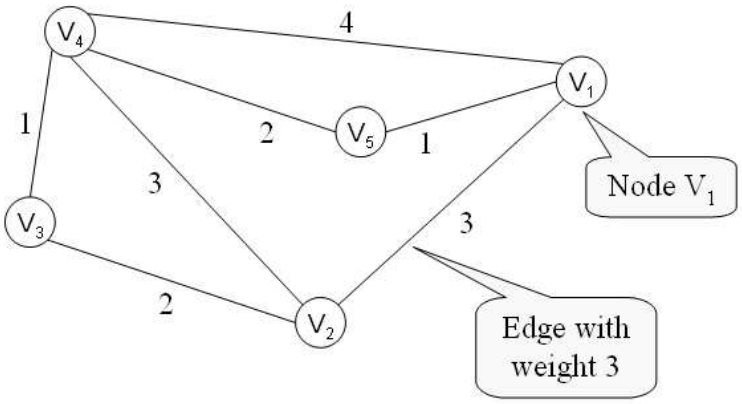

Figure 6.7 A weighted undirected Graph with 5 nodes and 7 edges

The task can be generalized to a general graph problem where a junction or city is called *node* or *vertex* and a connection of two nodes is called edge (see Figure 6.7). An edge has a positive number (called *weight*) attached representing the distance between the connecting nodes.

Looking for the shortest route from V_1 to V_4 we may start with the direct connection V_1 - V_4. As the "non-stop" route is not necessarily the shortest (having weight 4 in our example) because by avoiding to cross cities we may have to use a bypass that is longer (in distance) than crossing the city. That is why we have to check routes with stops as well.

To find the best route for given weights we could use combinatorics and try all possible ordered combination of nodes. In the general case we have to assume a fully connected graph, i.e. there is an edge for each combination of nodes. In this case we permutate all subsets of n nodes to find all possible routes. It is known from combinatorics that there are n factorial (notation: $n!$) ordered combination of nodes. These represent all possible routes via the n nodes. As we want to find all routes form a certain node (say V_1) to an other particular node (say V_4) our first and last node is fixed. Therefore only $n-2$ nodes are left to permutate. In addition we have to check routes with less than $n-2$ stops in between. In total there exist more than $(n-2)!$ routes in general for our example which shows that the number of routes to check is growing at factorial speed (this is even faster than exponential) with the number of nodes. Hence, the run time necessary for this “brute force” algorithm grows at a factorial magnitude with the number of nodes (cities). This is the reason why in practice it is not feasible to solve the shortest route problem with this straight forward approach for a larger number of nodes. Instead less resource consuming approximations are required.

There are many optimizations like the following. We can stop testing if a partial route is already longer than the shortest route found so far. But still we have to be aware that we may find an n-stop route to be shortest.

Below we presents two of the most prominent algorithms. First, we show an already classical, general algorithm for the *shortest path problem* with a

square time complexity. Then we present an heuristic search method, called A*-algorithm.

6.4.1 Dijkstra's Algorithm

Dijkstra's algorithm finds *shortest paths* in a network of positive weighted nodes between two connected nodes. It is possible to interpret the weight as travel time, travel costs, or as the value of a general *goal function*.

The algorithm starts from the source node, and successively adds another edge to the node set V that has a shortest path (sum of weights) to the starting node so far. The algorithm has to determine n times the next closest node out of a maximum of n nodes. Thus, the run time grows with the square order of the nodes in the network, i.e. the time complexity is $O(n^2)$.

The set V of visited nodes forms a growing tree with minimal distance from the source node (root) to the leaf nodes (see Figure 6.8. When a node is hit again we take the route with the shortest path (smallest sum of edge weights). Eventually all nodes are hit and we have found the shortest route to the destination node.

Let's illustrate the mechanism step-by-step with the following example. For simplicity we restrict our network to 7 nodes with 10 vertices. The distance between two nodes is indicated by the weight attached to an edge (Figure 6.8). We want to find the shortest route from node A to node G.

To prepare the search all nodes together with its distance to the starting node are put in a queue Q. As we don't know any distances yet we note infinity (∞) except for the source node A which has distance 0. Starting from node A the neighbor nodes B, E, and F are put into the set V (= visited node tree)and the distances for these nodes in the queue Q are adjusted. The starting node A is removed from the queue and node B as the closest to A (distance 2) will serve as starting point for the next search step (picture 1 of Figure 6.8). The nodes C and F are added to V and the distances are recalculated. Please note, that the distance from A to F is changed from 15 to 8 because the route via B is shorter than the "winding road" from A to F. The node B is removed from the queue Q. The closest node from A is now C with a distance of 6.

In the next iteration all nodes directly connected to node C are checked and new distances result for D and G. Now we have two nodes with distance 8 to the starting node A. We are free to choose either node D or F. We take D and improve the distance to G to 9.

Finally node E and G are left to search for shorter pathes. The next two iterations (picture 6 and 7 of Figure 6.8) do not change any distances because no shorter routes exist, it only empties the queue Q indicating that the algorithm has finished.

Finally we present Dijkstra's Algorithm more formally in Figure 6.4.1. The pseudo code uses the dot-notation *Graph.nodes* for the set of nodes and $dist(s, v)$ means the distance of node v to start node s. The priority queue PrioQ contains all nodes not processed yet in ascending order of distance to start node s. So that the next node to process is always the first element in the PrioQ. This node has the shortest distance to the start node so far. The weighted distance between two nodes u and v is denoted by *dist(v,u)*. The main operation is the *edge relaxation* if a shortest known path from node s to v is extended to its neighbor u resulting in the shortest known path from s to u. This path will have

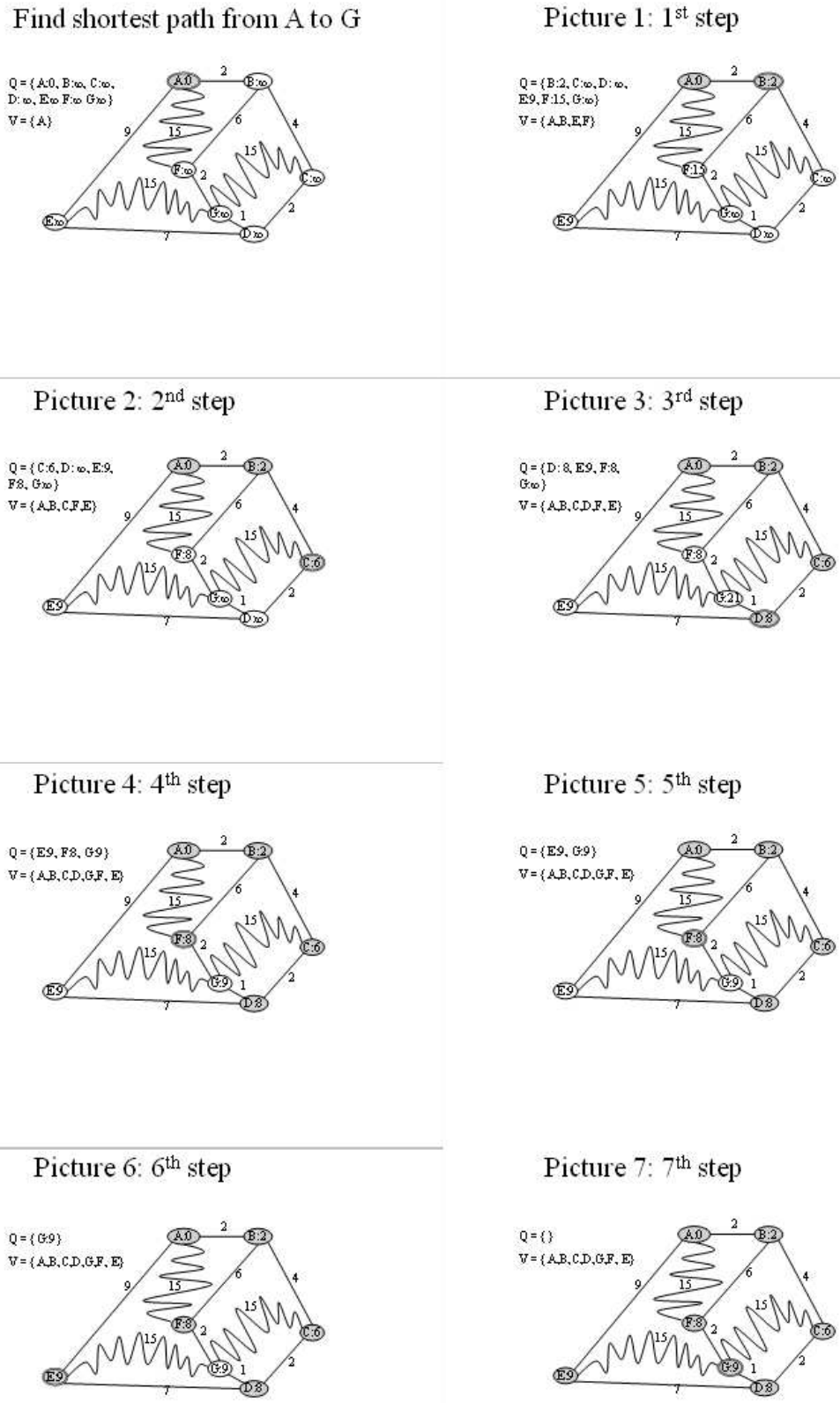

Figure 6.8 Sequence how Dijkstra's Algorithm works

```
function DijkstraAlgorithm(Graph, startNode)
  for each v ∈ Graph.nodes\s do
    dist(s, v) = ∞
  end for
  dist(s, s) := 0; PrioQ := Graph.nodes;
  while not PrioQ.isEmpty do
    v := PrioQ.NodeWithMinDist
    for each u ∈ v.neighbors ⋂ PrioQ do
      if (dist(s, v) + dist(v, u)) < dist(s, u) then
        dist(s,u) := dist(s,v) + dist(v,u) /* relax */
        reorder PrioQ with new dist(s,u) and remove processed nodes
      end if
    end for
  end while
  return dist(s,destNode)
end function
```

Figure 6.9 Pseudo code for Dijkstra's Algorithm for the shortest path problem

length $dist(s, v) + dist(v, u)$. If this is less than the current distance dist(s,u) we can replace the current value with the new value.

As we can see from the algorithm there is no information provided or assumption made in which "direction") or sequence to check the neighbors for continuing the shortest paths (the set $v.neigbors \bigcap PrioQ$ has no ordering). We call this an *uninformed* algorithm. This unawareness of the direction leads to many tries that do not contribute to the shortest paths. As example look at node F of Figure 6.8 which is directly connected to node A. This node is repeatedly checked in the priority queue and its distance to A is calculated two times. In the next section we will see how we could benefit from some heuristic information to prevent the computation of paths that are not promising.

6.4.2 A*-Algorithm

Dijkstra's algorithm is unaware of the most likely direction to move. If you want to travel from NY to LA and you have reached Oklahoma it is not promising to try a road to Chicago or New Orleans. It would be very useful to have some hint for a direction. In the real world of route planning a good hint for the direction to take is either the geographic direction or the next city that has the shortest beeline distance to the destination.

The A*-Algorithm uses a heuristic to estimate the distance from a node to the destination. If we take the route planning example the beeline distance could be used as heuristic. The beeline is a lower bound from any city to the destination. Assume that on the search for the shortest route you have visited some cities and you want to know which direction to check next. From all possible neighbor cities take that city where the sum of its beeline distance and the already traveled distance is a minimum. This is the most likely candidate for the shortest path. More formally the heuristic algorithm could be expressed as:

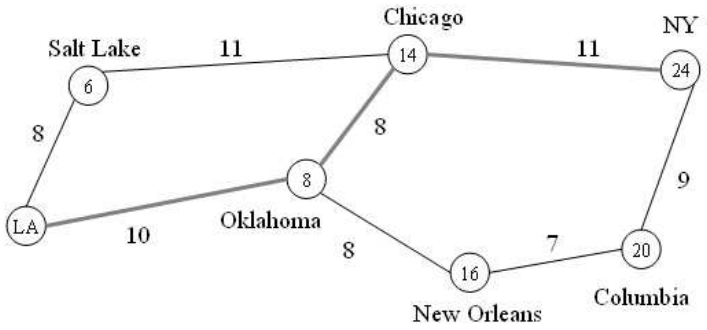

Figure 6.10 Finding shortest route from NY to LA (simplified map)

Let V be the set of visited cities. Build an ordered priority queue prioQ = $C_1, C_2, C_3, \ldots, C_n$ of neighbors cities where the cities are ordered with increasing estimated distance $f(C_i) = C_i.dist + h(C_i)$ from start to destination. The ordering ensures that $f(C_i) \leq f(C_k)$ for all i < k where $C_i.dist$ denotes the distance from the start city and $h(C_i)$ is the beeline distance (the estimated or heuristic value) to the destination.

We explain the concept for the A*-Algorithm with an example. Assume we want to travel by car from NY to LA. We know the beeline distance from each intermediate city to LA. Figure 6.10 shows a simplified scenario. The distance (in hundreds of miles) between two neighbor cities is written next to the edge connecting both cities (nodes). The beeline distance form a city to LA is written inside the node.

Starting from NY we have two neighbor cities. Chicago is in distance 11 units[3] and Columbia is 9 units away. As Chicago has a beeline distance to LA of 14, the sum is $25 = 11 + 14$ compared to $29 = 9 + 20$ via Columbia. Therefore, it is more promising to continue the route via Chicago and not via Columbia. Chicago and Columbia are added to prioQ. With each city in the prioQ the following attributes are associated: estimated distance $f(C_i)$, the traveled distance $C_i.dist$, and for simplicity the route r taken. We write these information as a triple $C_i(f, dist, r)$. At the moment we have traveled to Chicago and to Columbia which are put into the priority queue:

$$\text{prioQ} = \Big(\text{Chicago(25,11,NY)}, \text{Columbia(29,9,NY)}\Big).$$

Next to Chicago is NY, Salt Lake, and Oklahoma. We have now to visit these cities and reorder the priority list for NY, Salt Lake, Oklahoma, and Columbia. The route via Salt Lake has an estimated distance of $11 + 11 + h(\text{Salt Lake}) = 28$. For Oklahoma we get $11 + 8 + h(\text{Oklahoma}) = 27$, and for NY $11 + 11 + h(NY) = 46$. Now the priority Queue looks like

[3] all distances are in 100 miles

$$\text{prioQ} = \Big(\text{Oklahoma(27,19,NY}\rightarrow\text{Chicago), Salt Lake((28,22,NY}\rightarrow\text{Chicago),}$$
$$\text{Columbia(29,9,NY), NY(46,24,NY}\rightarrow\text{Chicago)}\Big)$$

Even if Salt Lake is closer to LA in beeline distance than Oklahoma we select Oklahoma as next city because the estimated total distance via Chicago and Oklahoma is shorter than that via Chicago and Salt Lake. The route via Columbia is also checked, but it is expected to be longer than the routes via Chicago. The route that visits NY again can be excluded as we don't want to travel in circles.

The next cities from Oklahoma are New Orleans, Chicago, LA. Clearly we have now found a route to LA, but it is necessary to check if it is the shortest. The real total distance to LA via Chicago and Oklahoma is 29. As the last estimation via Salt Lake was only 28 we have to check this route first. The priority Queue is reorganized and reads now

$$\text{prioQ} = \Big(\text{Salt Lake(28,22,NY}\rightarrow\text{Chicago), LA(29,29,NY}\rightarrow\text{Chicago}\rightarrow\text{Oklahoma),}$$
$$\text{Columbia(29,9,NY), New Orleans(32,16,NY}\rightarrow\text{Columbia)}\Big)$$

First we try the route via Salt Lake and end up in a real distance of 30 which is more than the estimate via Oklahoma. There is no need to check the route via Columbia as this is *at least* 29. Finally we have found two routes to LA where the one LA(29,29,NY→Chicago→Oklahoma->) is the shortest.

If the A*-Algorithm is compared with Dijkstra's Algorithm it can be noted that with A* we do not need to try all possible routes like the one via Columbia and New Orleans. This is because we have an estimate (heuristic) that makes it unlikely that this might be the shortest route. In fact, if we never overestimate the distance A* will result in the shortest route. This is clearly true for the beeline distance as any other connection has to be at least that long.

Starting at a certain city C the A*-Algorithm has to determine in the worst case the shortest path estimate for $n-2$ cities. This is to be repeated for each city on the way to the destination. In the worst case the processing time needed is $(n-1)(n-2) = O(n^2)$ computations of path estimates. If we assume that our heuristic does not overestimate the distance and the number of routes to check is limited by a constant c the complexity is reduced to $(n-1)c = O(n)$. This situation will be found when a new route segment never increases the estimated distance so that it exceeds more than c previous estimates. In the example of Figure 6.10 this number c was only 2.

6.5 Histograms

For analyzing large amount of data it is often useful to group the values into categories. With this method the frequency distribution of values in the categories can be easily seen. In this sense histograms are a kind of approximation because you do not know the exact value of a single item but you know about the number of items within a certain value range.

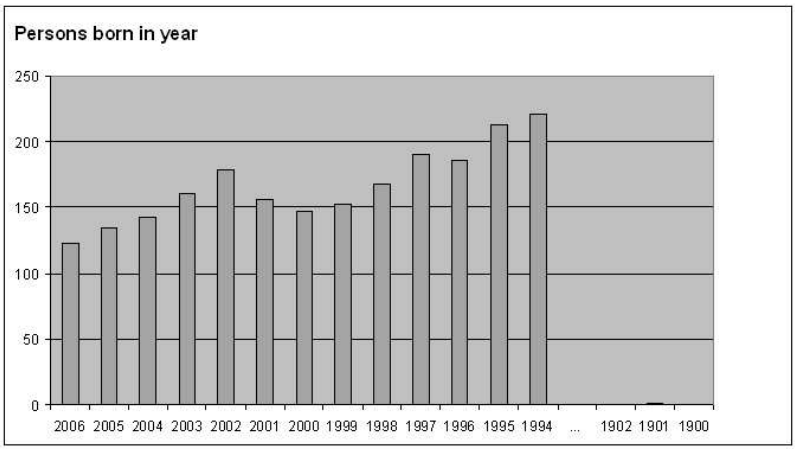

Figure 6.11 Histogram showing persons born in years 2006 ... 1900

As example take the age of people. We can group the persons according to their birth year into categories. All persons born in the same year belong to the same category and we only have to count the number of persons in each category to know how many people are born in a certain year. However, we do not know the birthday of a particular person.

The result is a table with two columns, the category (birth year) and the number of persons in that category.

birth year	number of persons
2006	123
2005	135
2004	142
2003	161
2002	179
2001	156
2000	147
1999	153
1998	168
1997	190
1996	186
1995	213
1994	221
...	...
1902	0
1901	1
1900	0

The corresponding histogram is shown in figure 6.11 visualizing much better than the table the trend for a decreasing people's birth rate.

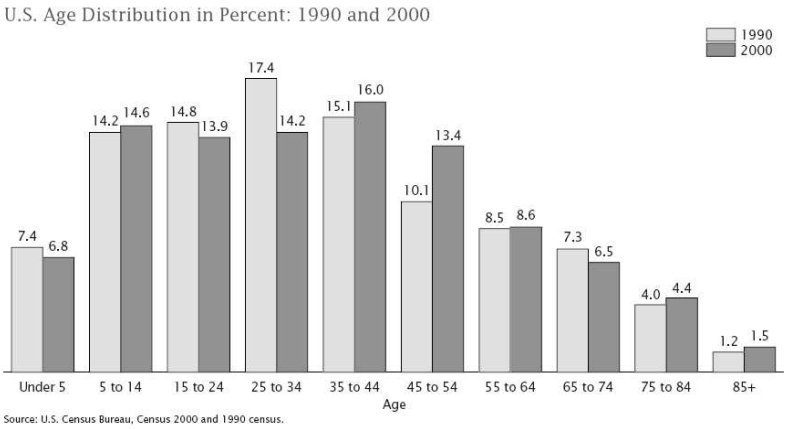

Figure 6.12 US age distribution in percent

A common representation of such a table is a bar chart where each bar represents a category and the hight represents the number of objects that belong to this category.

A **histogram** is a graphical mean to represent the frequency distribution of data values. Histograms help to easily overlook or statistically analyze quantitative or categorial data.

An *one dimensional histogram* is a discrete step function h where $y_i := f(t_i(i = 1, 2, ..., n)$ represent a distribution of objects in n classes. For each class i the variable t_i is the width of the class and y_i is the number of objects in that class. The function h can be written as a finite sequence of pairs (t_i, y_i) for $i = 1, 2, ..., n$.

For simple histograms all categories have the same "width" or range. This makes two categories comparable with regard to its frequencies. The range of a category can freely be chosen but it also determines the "resolution" of the histogram. In our example on the age of people we could have decided as category covering 10 years instead of one, but then we would have lost information about the age distribution within sub-decades. More than one distribution can be shown with the same categorizing scheme for better comparison. This is the case in Figure 6.12 where the US age distribution is visualized by histograms for the years 1990 and 2000 with the same categorization.

If the histogram function is drawn on a cartesian coordinate system each pair represents a bin of width t_i and hight y_i like in Figure 6.11. The classes are listed on the abscissa in its sequence $1, 2, ..., n$. Then the rectangular shapes for the bins look like a list of vertical bars or "masts". This explains the origin of the name quotedhistogram as from the Greek 'isto-s' ($\iota\sigma\tau o\sigma$) = 'mast', and 'gram-ma' ($\gamma\rho\alpha\mu\mu\alpha$) (= 'something written', 'a symbol').

Depending on the categorizing bins and the scale of the histogram the results may appear very different. This opens the possibility to use histograms to mislead the naive reader toward conclusions which are not really as indicated.

Here is an example for a misuse of an histogram. A motor company reports the following car breakdown percentages:

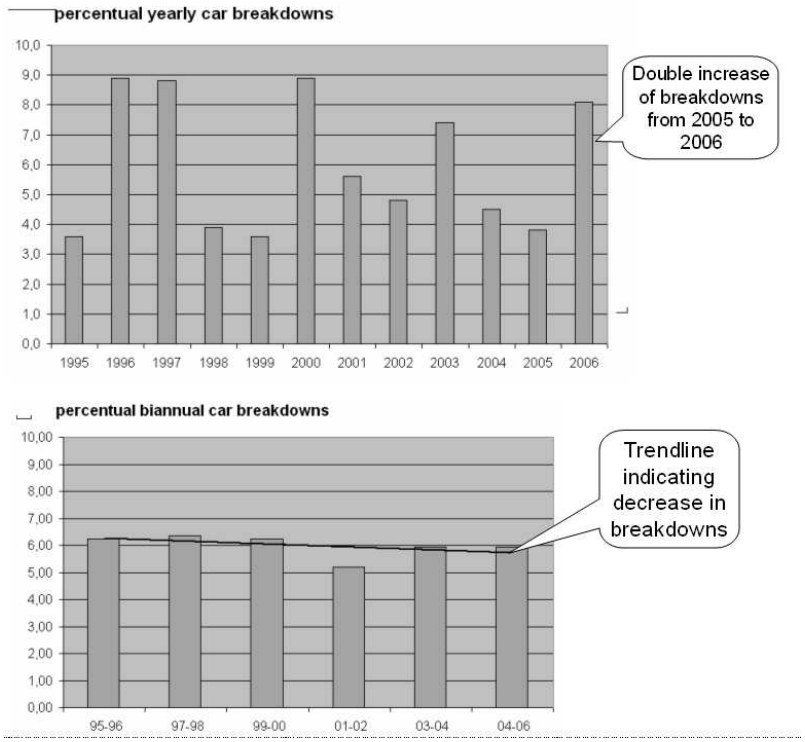

Figure 6.13 Percentage of car breakdowns

year	breakdown percentage
1995	3.6
1996	8.9
1997	8.8
1998	3.9
1999	3.6
2000	8.9
2001	5.6
2002	4.8
2003	7.4
2004	4.5
2005	3.8
2006	8.1

The motor company is claiming that their cars face less breakdowns in the last years and that the rate is below 7 percent. How would you group the numbers that the histogram can support this statement? Figure 6.13 shows a possibility to support the claim. Please note that the breakdowns in 2006 have actually raised substantially compared to the years 2004 and 2005. Nevertheless the trend line indicates a slight (non-significant) decrease in breakdowns. This is a clear example how statistics and histograms in particular can be "legally" misleading if used unsoundly.

6.6 Time-accuracy Tradeoff

to do
What is that?

- numerical methods for computing ?
- iterations towards a solution (fix point, error bound)?
- error propagation ?

6.7 Summary

Exercises

6.1 Write down a heuristic for how to read a lecture book if you only want to know about a certain topic.

6.2 Develop an algorithm for finding prime numbers using the dynamic programming principle.

6.3 Write a program with a MiniMax algorithm for the Tic-Tac-Toe game.

6.4 Find a animated visualization of the A*-Algorithm.

6.5 How could you show with an histogram which consumer groups buy how many computers over time?

Bibliographical Notes

George Polya wrote a remarkable book **?** on problem solving techniques that is useful not only for the Computer Science realm but for problems in general.

The eminent computer scientist and Turing award winner Edsger Dijkstra published in the year 1959 a shortest path algorithm of square order complexity, i.e. $O(n^2)$. This algorithm was the seed for refined, extended, or heuristic variants.