2.6 Advanced Algorithms: Searching, Sorting and Path-Finding

OCR H446 · Calculators not allowed · about 120 minutes · Write-in edition

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 A satnav app calculates driving routes across a national road network containing millions of junctions.

(a) Discuss the relative advantages of using Dijkstra's algorithm compared with the A* algorithm for calculating a single driving route between two specific junctions in this network, and describe one situation in which Dijkstra's algorithm would still be the more appropriate choice. **(8)**

(Total for Question 1 is 8 marks)

2 An admissible heuristic h(v) for a search from vertex v to a goal G never overestimates the true shortest remaining distance from v to G, i.e. h(v) ≤ actualDistance(v, G) always. A consistent (monotone) heuristic additionally satisfies h(u) ≤ cost(u, v) + h(v) for every edge (u, v). A* is only guaranteed to find the shortest path when its heuristic is admissible.

(a) In the courier network used earlier in this paper, the true shortest remaining journey time from D to F is 5 minutes, and from E to F is 3 minutes. State, with a reason, whether the heuristic values h(D) = 4 and h(E) = 2 used earlier are each admissible. **(2)**

(b) Show that the heuristic values h(D) = 4 and h(E) = 2 satisfy the consistency condition for the edge D-E (weight 2). **(2)**

(c) A much simpler graph has three vertices: S (start), X, and G (goal), plus a fourth vertex Y. Edges: S-X weight 1, X-G weight 5, S-Y weight 1, Y-G weight 1. The true shortest distance from S to G is therefore 2 (via Y: S-Y-G), not 6 (via X: S-X-G). Suppose A* is run using the heuristic values h(X) = 0 and h(Y) = 10 (note: h(Y) = 10 is NOT admissible, since the true remaining distance from Y to G is only 1). Complete the trace below to show which vertex A* expands at each step, and state the path and total distance A* returns. **(6)**

step	vertex expanded	g, f of vertices reached
1	S	X: g=1, f=1 ; Y: g=1, f=11
2		
3		

(d) State what this example demonstrates about the importance of the admissibility condition for A* to guarantee an optimal (shortest) result. **(1)**

(Total for Question 2 is 11 marks)

3 A warehouse robot navigates a grid-shaped map with R rows and C columns of cells. Each cell is connected to its (up to 4) directly adjacent cells (up, down, left, right), forming a graph where each cell is a vertex.

(a) State, in terms of R and C, the number of vertices V in this graph. **(1)**

(b) Show that the number of edges E in this grid graph is given by E = R(C - 1) + C(R - 1). **(2)**

(c) The warehouse grid has R = 100 and C = 100 (a 100 x 100 grid). Calculate V and E for this grid. **(3)**

(d) Using the adjacency-list-with-min-heap implementation of Dijkstra/A*, with time complexity O((V + E) log V), calculate an estimate for the number of operations needed to find a route across this 100 x 100 grid. Give your answer to 3 significant figures. **(3)**

(e) In practice, A* with a well-chosen heuristic (such as Manhattan distance, |x1 - x2| + |y1 - y2|, between grid coordinates) typically explores far fewer than V vertices when finding a route between two specific cells, even though the algorithm's worst-case time complexity calculated in part (d) is the same as Dijkstra's. Explain why this is the case. **(2)**

(Total for Question 3 is 11 marks)

4 The maximum number of key comparisons needed by merge sort to fully sort a list of n elements (where n is a power of 2) satisfies the recurrence relation:

$C(n) = 2 \times C(n / 2) + (n - 1)$, with $C(1) = 0$

This reflects recursively sorting two half-lists, each needing C(n/2) comparisons, and then merging the two sorted halves together using at most (n - 1) comparisons.

(a) Use the recurrence relation to calculate C(2), C(4) and C(8), showing your working at each stage. **(3)**

(b) It can be shown that, for n a power of 2, $C(n) = n \times \log2(n) - n + 1$. Verify that this formula gives the same value for C(8) as your answer to part (a). **(2)**

(c) Use the closed-form formula from part (b) to calculate C(16) and C(64), without using the recurrence relation directly. **(4)**

(d) A list of 64 elements is sorted using merge sort, then a second list of 64 elements is sorted using an $O(n^2)$ algorithm such as bubble sort, which needs up to $n(n - 1) / 2$ comparisons in the worst case. Calculate this worst-case number of comparisons for $n = 64$, and comment on what the comparison with your answer for $C(64)$ shows about the relative growth rates of $O(n \log n)$ and $O(n^2)$. **(3)**

(Total for Question 4 is 12 marks)

5 The list [8, 3, 10, 1, 6, 14, 4] is sorted using the version of quicksort below, which always chooses the first element of the current list as the pivot.

```
FUNCTION QuickSort(list) RETURNS ARRAY
  IF LEN(list) <= 1 THEN
    RETURN list
  ENDIF
  pivot <- list[0]
  less <- []
  greaterEq <- []
  FOR i <- 1 TO LEN(list) - 1
    IF list[i] < pivot THEN
      APPEND(less, list[i])
    ELSE
      APPEND(greaterEq, list[i])
    ENDIF
  NEXT i
  RETURN QuickSort(less) + [pivot] + QuickSort(greaterEq)
ENDFUNCTION
```

(a) State the pivot used, and the resulting less and greaterEq lists, for the first call QuickSort([8, 3, 10, 1, 6, 14, 4]). **(3)**

(b) The algorithm then recursively calls QuickSort(less) and QuickSort(greaterEq). Complete the table below giving the pivot and the resulting less and greaterEq lists produced at each further recursive call on a list of more than one element. **(4)**

call	input list	pivot	less	greaterEq
2	[3, 1, 6, 4]			
3	[6, 4]			
4	[10, 14]			

(c) State the final sorted list returned by QuickSort([8, 3, 10, 1, 6, 14, 4]), and calculate the total number of element-to-pivot comparisons made across the whole sort (i.e. summing the comparisons made in every recursive call). **(3)**

(d) State the average-case time complexity of quicksort, and explain briefly why this typically holds when, as in this example, the pivot splits the list into two roughly similar-sized parts at each level of recursion. **(2)**

(Total for Question 5 is 12 marks)

6 The same QuickSort algorithm from the previous question (pivot always chosen as list[0]) is now applied to a list that is already sorted in ascending order: [2, 5, 7, 9, 12, 15] (6 elements).

(a) State the pivot, and the resulting less and greaterEq lists, for the first call QuickSort([2, 5, 7, 9, 12, 15]). **(2)**

(b) Explain why, for this sorted input with this pivot-choice strategy, every recursive call will produce an empty less list and a greaterEq list containing all of the remaining elements. **(2)**

(c) Complete the table below showing the number of element-to-pivot comparisons made at each level of recursion, continuing until the list is fully sorted, and state the total number of comparisons made overall. **(3)**

level	list size at this level	comparisons made
1	6	5
2	5	
3	4	
4	3	
5	2	
6	1	

(d) Show that, in general, this worst-case total number of comparisons for a sorted list of n elements is n(n - 1) / 2, and state the resulting Big O time complexity. **(3)**

(e) State two different strategies for choosing the pivot that would make this particular worst-case scenario (an already-sorted list) far less likely to occur in practice. **(2)**

(Total for Question 6 is 12 marks)

7 A courier company's road network is modelled as a weighted graph. Vertices A to F represent depots, and edge weights represent journey times in minutes between directly connected depots:

Edges: A-B: 4, A-C: 2, B-C: 1, B-D: 5, C-D: 8, C-E: 10, D-E: 2, D-F: 6, E-F: 3

Dijkstra's algorithm is used to find the shortest journey time from depot A to every other depot.

(a) State one condition that all edge weights in a graph must satisfy for Dijkstra's algorithm to correctly find shortest paths. **(1)**

(b) Complete the table below to trace Dijkstra's algorithm starting from A. At each step, give the vertex settled and the current shortest known distance to each vertex (write "-" if a distance is not yet known). The first row is completed for you. **(5)**

step	vertex settled	A	B	C	D	E	F
1	A	0	4	2	-	-	-
2							
3							
4							
5							
6							

(c) State the shortest journey time from A to F, and give the corresponding path of depots visited. **(2)**

(d) Explain how the path found in part (c) can be reconstructed from the completed table, without the algorithm needing to store the whole path directly as it runs. **(2)**

(Total for Question 7 is 10 marks)

8 Dijkstra's algorithm can be implemented in different ways, giving different overall time complexities. Let V be the number of vertices and E the number of edges in the graph.

(a) State the overall time complexity of Dijkstra's algorithm when implemented using an adjacency matrix, where finding the next unvisited vertex with the smallest tentative distance requires scanning all V vertices, and this scan is repeated once for each of the V vertices settled. **(2)**

(b) State the overall time complexity of Dijkstra's algorithm when implemented using an adjacency list together with a min-heap (priority queue) to select the next vertex, where each of the E edges may cause one O(log V) update to the heap. **(2)**

(c) The courier network in the previous question has V = 6 vertices and E = 9 edges. Calculate the approximate number of operations each implementation from parts (a) and (b) would require. **(4)**

(d) State, with a reason, which implementation would be expected to show a clearer advantage as the size of the network grows very large, assuming the network stays sparse (E grows roughly in proportion to V, rather than to V^2). **(2)**

(Total for Question 8 is 10 marks)

9 The same courier network from the previous two questions is now searched using the A* algorithm to find the shortest journey time from A to F. Straight-line distance estimates (heuristic values) to F are given below, and are known to be admissible (never greater than the true remaining journey time to F):

h(A) = 9, h(B) = 7, h(C) = 8, h(D) = 4, h(E) = 2, h(F) = 0

At each step, A* expands the vertex in the open list with the smallest value of f(vertex) = g(vertex) + h(vertex), where g(vertex) is the shortest distance found so far from A to that vertex.

(a) Calculate f(A) at the start of the search, given g(A) = 0. **(1)**

(b) A is expanded first. Calculate g, h and f for its neighbours B and C once A has been expanded. **(2)**

(c) Complete the table below to continue the A* trace, showing which vertex is expanded at each step and the resulting g and f values of any vertices whose distance is improved as a result. The first two steps are shown. **(4)**

step	vertex expanded	updated vertices (g, f)
1	A	B: g=4, f=11 ; C: g=2, f=10
2	C	B: g=3, f=10 ; D: g=10, f=14 ; E: g=12, f=14
3		
4		
5		
6		

(d) State the total journey time and the path found by A* from A to F, and compare this result with the result obtained using Dijkstra's algorithm in the earlier question. **(3)**

(e) State the total number of vertices expanded by A* in this trace, and comment on whether this shows an efficiency advantage over Dijkstra's algorithm for this particular graph. **(3)**

(Total for Question 9 is 13 marks)

Mark scheme · 2.6 Advanced Algorithms: Searching, Sorting and Path-Finding

Question 1

(a) **L1** Basic, isolated points are made about Dijkstra's algorithm and/or A*, with little or no reference to time complexity, heuristics or the given scenario. The answer may simply name both algorithms with little justification, or fail to identify any valid situation where Dijkstra is preferable. **1-3**

(a) **L2** Some relevant points are made about the efficiency of A* compared with Dijkstra, with limited or partly-accurate reference to heuristics and time complexity. Some link is made to the scale of the scenario (millions of junctions), and a situation favouring Dijkstra is attempted but may be only partially justified. **4-6**

(a) **L3** A clear, well-justified discussion is given, correctly explaining how A*'s heuristic reduces the number of junctions explored compared with Dijkstra for a single specific route, supported by accurate technical detail (e.g. admissibility). A valid, clearly justified situation where Dijkstra remains the more appropriate choice is identified and linked to the scenario. **7-8**

(a) Indicative content:

- A* uses a heuristic (e.g. straight-line distance to the destination) to prioritise exploring junctions likely to lie on the route to the specific destination, so across a network of millions of junctions it can typically reach the destination while expanding far fewer vertices than Dijkstra's algorithm, which explores outward in all directions with no sense of direction towards the goal.
- Both algorithms are guaranteed to find the shortest path provided the heuristic used by A* is admissible (never overestimates the true remaining distance); straight-line distance is always less than or equal to the true road distance, so it is a valid admissible heuristic here.
- Dijkstra's algorithm remains the more appropriate choice when the shortest paths from a single junction to every other junction are needed at once (a single-source, all-destinations problem) - for example, precomputing journey times from a delivery depot to every possible drop-off point overnight - because A*'s heuristic advantage only helps when searching towards one specific, known destination.
- Dijkstra may also be preferred when no reliable, cheap-to-compute heuristic is available for the scenario, since a poor heuristic could remove most or all of A*'s efficiency benefit (in the extreme case, a heuristic that is always 0 makes A* behave identically to Dijkstra).
- A stronger answer may note the practical trade-off: A* needs a suitable heuristic (e.g. precomputed junction coordinates) available for every vertex, which adds a small implementation and data overhead that plain Dijkstra does not require.

(a) Answer: Best answers recommend A* for a single specific route, using an admissible heuristic such as straight-line distance to prioritise the search and expand far fewer junctions than Dijkstra, but correctly identify that Dijkstra is preferable when shortest paths to all destinations are needed simultaneously (e.g. depot-to-everywhere precomputation), since A*'s heuristic gives no benefit in that case.

Question 2

(a) h(D) = 4 is admissible because 4 ≤ 5 (the true shortest distance from D to F) **B1**

(a) h(E) = 2 is admissible because 2 ≤ 3 (the true shortest distance from E to F) **B1**

(a) Answer: Both are admissible: h(D)=4 ≤ 5, and h(E)=2 ≤ 3.

(b) checks h(D) ≤ cost(D, E) + h(E): 4 ≤ 2 + 2 **M1**

(b) 4 ≤ 4 is true (holds, with equality), so the heuristic is consistent for this edge cao **A1**

(b) Answer: h(D) = 4 ≤ cost(D,E) + h(E) = 2 + 2 = 4, so the condition holds (with equality).

(c) step 2: expands X (lowest f = 1, since f(Y) = 11) **B1**

(c) step 2: G is reached via X with g = 1 + 5 = 6, f = 6 + 0 = 6 **B1**

(c) step 3: expands G (f = 6, now the lowest in the open list, since f(Y) = 11) **B1**

(c) G is the goal, so A* terminates, returning the path S-X-G with total distance 6 **B1**

(c) recognises this is NOT the true shortest path, which is S-Y-G with total distance 2 **B1**

(c) recognises Y was never expanded before the search terminated **B1**

(c) Answer: Step2: expand X, G reached with g=6, f=6. Step3: expand G (lowest f=6, since f(Y)=11), goal reached, A* returns path S-X-G, distance 6. This is not the true shortest path (S-Y-G, distance 2); Y was never expanded.

(d) if a heuristic overestimates the true remaining distance for even one vertex, A* can terminate having found a suboptimal path, because that vertex is wrongly deprioritised (given a high f value) and so is never explored before the algorithm stops; an admissible heuristic is therefore essential for A*'s optimality guarantee **B1**

(d) Answer: It shows that an inadmissible (overestimating) heuristic can cause A* to terminate with a suboptimal path, because the vertex leading to the true shortest path is wrongly given a low priority and never explored before the search stops.

Question 3

(a) V = R x C **B1**

(a) Answer: V = R x C

(b) each of the R rows contains (C - 1) adjacent horizontal pairs of cells, giving R(C - 1) horizontal edges in total **M1**

(b) each of the C columns contains (R - 1) adjacent vertical pairs of cells, giving C(R - 1) vertical edges in total; adding these gives E = R(C - 1) + C(R - 1) **A1**

(b) Answer: E = R(C - 1) + C(R - 1)

(c) V = 100 x 100 **M1**

(c) E = 100 x 99 + 100 x 99 **M1**

(c) V = 10 000, E = 19 800 cao **A1**

(c) Answer: V = 10,000; E = 19,800

(d) V + E = 10 000 + 19 800 = 29 800 **M1**

(d) log2(10 000) = awrt 13.3 **M1**

(d) 29 800 x 13.3 = awrt 396 000 (accept 390 000 to 400 000) cao **A1**

(d) Answer: Approximately 396,000 operations.

(e) the heuristic gives the algorithm a strong sense of direction towards the specific goal cell, so it prioritises expanding cells that lie roughly between the start and the goal, rather than exploring in every direction **B1**

(e) as a result, in typical (non-worst-case) situations A* settles only a small fraction of the total V cells before reaching the goal, even though the stated worst-case bound O((V + E) log V) - which assumes the heuristic gives little useful guidance - remains identical to Dijkstra's (oe) **B1**

(e) Answer: The Manhattan-distance heuristic directs A* towards cells lying between the start and goal rather than in every direction, so in typical cases it expands far fewer than V cells before reaching the goal, even though its worst-case bound is the same as Dijkstra's.

Question 4

(a) C(2) = 2 x C(1) + (2 - 1) = 2 x 0 + 1 = 1 **M1**

(a) C(4) = 2 x C(2) + (4 - 1) = 2 x 1 + 3 = 5 **M1**

(a) C(8) = 2 x C(4) + (8 - 1) = 2 x 5 + 7 = 17 cao **A1**

(a) Answer: C(2) = 1, C(4) = 5, C(8) = 17

(b) substitutes n = 8: C(8) = 8 x log2(8) - 8 + 1 = 8 x 3 - 8 + 1 **M1**

(b) = 17, matching part (a) cao **A1**

(b) Answer: C(8) = 8 x 3 - 8 + 1 = 17, matching part (a).

(c) log2(16) = 4, so C(16) = 16 x 4 - 16 + 1 **M1**

(c) C(16) = 49 cao **A1**

(c) log2(64) = 6, so C(64) = 64 x 6 - 64 + 1 **M1**

(c) C(64) = 321 cao **A1**

(c) Answer: C(16) = 49, C(64) = 321

(d) 64 x 63 / 2 **M1**

(d) = 2016 cao **A1**

(d) valid comment: bubble sort needs more than six times as many comparisons as merge sort (2016 compared with 321) at this size, showing $O(n^2)$ grows substantially faster than O(n log n) as n increases (oe) **B1**

(d) Answer: 2016 comparisons for bubble sort, compared with 321 for merge sort; $O(n^2)$ clearly grows much faster than O(n log n) as n increases.

Question 5

(a) pivot = 8 **B1**

(a) less = [3, 1, 6, 4] **B1**

(a) greaterEq = [10, 14] **B1**

(a) Answer: pivot = 8; less = [3, 1, 6, 4]; greaterEq = [10, 14]

(b) call 2 ([3,1,6,4]): pivot = 3, less = [1], greaterEq = [6, 4] **B1**

(b) call 3 ([6,4]): pivot = 6, less = [4], greaterEq = [] **B1**

(b) call 4 ([10,14]): pivot = 10, less = [], greaterEq = [14] **B1**

(b) correctly recognises that lists of 0 or 1 elements (e.g. [1], [4], [14], []) are base cases and are returned immediately with no further partitioning **B1**

(b) Answer: Call2 ([3,1,6,4]): pivot=3, less=[1], greaterEq=[6,4]. Call3 ([6,4]): pivot=6, less=[4], greaterEq=[]. Call4 ([10,14]): pivot=10, less=[], greaterEq=[14].

(c) [1, 3, 4, 6, 8, 10, 14] ft **B1**

(c) 6 (call 1) + 3 (call 2) + 1 (call 3) + 1 (call 4) shown **M1**

(c) 11 comparisons cao **A1**

(c) Answer: [1, 3, 4, 6, 8, 10, 14]; 11 comparisons in total.

(d) O(n log n) cao **B1**

(d) when partitions are roughly balanced, the list size roughly halves at each level, giving about log2(n) levels of recursion, and each level does a total of about n comparisons across all its calls, giving n x log n overall (oe) **B1**

(d) Answer: O(n log n): with roughly balanced partitions, there are about log2(n) levels of recursion, and each level does a total of about n comparisons, giving n x log n overall.

Question 6

(a) pivot = 2, less = [] **B1**

(a) greaterEq = [5, 7, 9, 12, 15] **B1**

(a) Answer: pivot = 2; less = []; greaterEq = [5, 7, 9, 12, 15]

(b) the pivot chosen (list[0]) is always the smallest of the remaining elements, since the list is sorted in ascending order **B1**

(b) so every other remaining element is greater than or equal to the pivot, meaning nothing can ever go into less, and the recursion becomes a single, highly unbalanced chain rather than two roughly equal halves (oe) **B1**

(b) Answer: Because the list is sorted, the first element (chosen as pivot) is always the smallest of what remains, so every other element is greater than or equal to it; this means less is always empty and greaterEq always contains everything else, giving one very unbalanced recursive branch instead of two balanced halves.

(c) levels 2-6: comparisons = 4, 3, 2, 1, 0 respectively **B1**

(c) recognises the pattern: comparisons made equal (list size at that level) - 1 **B1**

(c) total = 5 + 4 + 3 + 2 + 1 + 0 = 15 cao **B1**

(c) Answer: Comparisons per level: 5, 4, 3, 2, 1, 0. Total = 15.

(d) total comparisons = (n - 1) + (n - 2) + ... + 1 + 0, the sum of the first (n - 1) positive integers **M1**

(d) this arithmetic series sums to n(n - 1) / 2; verifying for n = 6 gives 6 x 5 / 2 = 15, matching part (c) **A1**

(d) $O(n^2)$ **B1**

(d) Answer: Total comparisons = n(n - 1) / 2, which is $O(n^2)$. For n = 6 this gives 15, matching part (c).

(e) choosing the pivot at random from the current list **B1**

(e) choosing the pivot as the median of the first, middle and last elements ("median-of-three"), or equivalently choosing the middle element of the list **B1**

(e) Answer: Choosing the pivot at random, or choosing it as the median-of-three (or middle element) of the current list.

Question 7

(a) all edge weights must be non-negative (zero or positive) **B1**

(a) Answer: All edge weights must be non-negative.

(b) step 2: settles C; distances A=0, B=3, C=2, D=10, E=12, F=- **B1**

(b) step 3: settles B; distances A=0, B=3, C=2, D=8, E=12, F=- **B1**

(b) step 4: settles D; distances A=0, B=3, C=2, D=8, E=10, F=14 **B1**

(b) step 5: settles E; distances A=0, B=3, C=2, D=8, E=10, F=13 **B1**

(b) step 6: settles F; distances unchanged, F=13 **B1**

(b) Answer: Step2: settle C (B=3,C=2,D=10,E=12). Step3: settle B (D=8). Step4: settle D (E=10,F=14). Step5: settle E (F=13). Step6: settle F (13).

(c) 13 minutes ft **B1**

(c) A-C-B-D-E-F ft **B1**

(c) Answer: 13 minutes, via the path A-C-B-D-E-F.

(d) for each vertex, the algorithm can record which neighbouring vertex it was reached from at the moment its shortest distance was last updated (its "predecessor") **B1**

(d) starting at the destination, following the predecessor links backwards until reaching the source gives the path, in reverse order **B1**

(d) Answer: By storing, for every vertex, the predecessor it was reached from when its shortest distance was last set; starting at F and following predecessors back to A reconstructs the path in reverse.

Question 8

(a) $O(V^2)$ **B1**

(a) V vertices are settled, and each requires an O(V) scan to find the next closest unvisited vertex, giving V x V = V^2 (oe) **B1**

(a) Answer: $O(V^2)$: V vertices are settled, each needing an O(V) scan, giving V x V operations.

(b) O((V + E) log V) **B1**

(b) extracting the minimum V times costs O(V log V), and each of the E edges can trigger one O(log V) heap update, giving O((V + E) log V) overall (oe) **B1**

(b) Answer: O((V + E) log V): V extractions each cost O(log V), and E possible heap updates each cost O(log V).

(c) adjacency matrix: $V^2 = 6^2$ **M1**

(c) = 36 cao **A1**

(c) adjacency list + heap: (V + E) x log2(V) = (6 + 9) x log2(6) = 15 x awrt 2.585 **M1**

(c) awrt 39 (accept 38-39) **A1**

(c) Answer: Adjacency matrix: approximately 36 operations. Adjacency list + heap: approximately 39 operations.

(d) the adjacency list with min-heap implementation **B1**

(d) because in a sparse graph E grows only linearly with V, so (V + E) log V grows far more slowly than V^2 as V becomes large, meaning the matrix implementation's V^2 term comes to dominate for large sparse graphs (oe) **B1**

(d) Answer: The adjacency list with min-heap, because for a sparse, large graph (V + E) log V grows far more slowly than V^2, so the matrix approach becomes comparatively far less efficient as V grows.

Question 9

(a) f(A) = 0 + 9 = 9 **B1**

(a) Answer: f(A) = 9

(b) B: g = 0 + 4 = 4, h = 7, f = 11 **B1**

(b) C: g = 0 + 2 = 2, h = 8, f = 10 **B1**

(b) Answer: B: g=4, h=7, f=11. C: g=2, h=8, f=10.

(c) step 3: expand B (lowest f = 10); D updates to g=8, f=12 **B1**

(c) step 4: expand D (lowest f = 12); E updates to g=10, f=12; F: g=14, f=14 (newly reached) **B1**

(c) step 5: expand E (lowest f = 12); F updates to g=13, f=13 **B1**

(c) step 6: expand F (lowest f = 13); F is the goal, so the search stops **B1**

(c) Answer: Step3: expand B, D updates g8/f12. Step4: expand D, E updates g10/f12, F newly g14/f14. Step5: expand E, F updates g13/f13. Step6: expand F, goal reached, stop.

(d) 13 minutes ft **B1**

(d) A-C-B-D-E-F ft **B1**

(d) this is identical to the distance and path found by Dijkstra's algorithm, confirming both algorithms found the true shortest path **B1**

(d) Answer: 13 minutes via A-C-B-D-E-F, identical to the result found by Dijkstra's algorithm.

(e) 6 vertices expanded (A, C, B, D, E, F), the same number settled by Dijkstra's algorithm **B1**

(e) no efficiency advantage is shown in this particular example, as both algorithms expanded/settled all 6 vertices in the same order **B1**

(e) valid reason: because F lies at the far end of this small, densely connected graph, almost every vertex lies on or close to the shortest path towards it, giving the heuristic little opportunity to avoid expanding any vertex; A*'s advantage is typically far greater on larger graphs, or when many vertices lie well away from the goal (oe) **B1**

(e) Answer: 6 vertices were expanded, exactly the same as Dijkstra settled, in the same order; no advantage is shown here because F lies at the far end of this small graph, so nearly every vertex lies on a shortest path towards it. A*'s benefit is normally much clearer on larger graphs.