۱. [۲۷نمره] درست یا نادرست بودن سوالات زیر را همراه با دلیل مشخص کنید. ⋆

(a) **F** In a weighted undirected graph $G = (V, E, w)$, breadth-first search from a vertex s finds single-source shortest paths from s (via parent pointers) in $O(V + E)$ time.

False. Only in unweighted graphs

(b) **T** In a weighted undirected tree $G = (V, E, w)$, breadth-first search from a vertex s finds single-source shortest paths from s (via parent pointers) in $O(V + E)$ time.

True. In a tree, there is only one path between two vertices, and breadth-first search finds it.

(c) **T** In a weighted undirected tree $G = (V, E, w)$, depth-first search from a vertex s finds single-source shortest paths from s (via parent pointers) in $O(V + E)$ time.

True. In a tree, there is only one path between two vertices, and depth-first search finds it.

(d) **F** If a graph represents courses and their prerequisites (i.e., an edge (u, v) indicates that course u must be taken before course v), then the breadth-first search order of vertices is a valid order in which to pass the courses.

No, you'd prefer depth-first search, which can easily be used to produce a topological sort of the graph, which would correspond to a valid course order. BFS can produce in correct results.

(e) **F** Dijkstra's shortest-path algorithm may relax an edge more than once in a graph with a cycle.

False. Dijkstra's algorithm **always** visits each node at most once; this is why it produces an incorrect result in the presence of negative-weight edges.

(f) **F** Given a weighted directed graph $G = (V, E, w)$ and a source $s \in V$, if G has a negative-weight cycle somewhere, then the Bellman-Ford algorithm will necessarily compute an incorrect result for some $\delta(s, v)$.

False. The negative-weight cycle has to be reachable from s.

(g) **F** In a weighted directed graph $G = (V, E, w)$ containing a negative-weight cycle, running the Bellman-Ford algorithm from s will find a shortest acyclic path from s to a given destination vertex t.

False. Bellman-Ford will terminate, and can detect the presence of that negative-weight cycle, but it can't "route around it." (You could always remove an edge to break the cycle and try again, though.)

(h) **F** The bidirectional Dijkstra algorithm runs faster than the Dijkstra algorithm even in a worst-case.

False.The constant factor behind bidirectional Dijkstra is better, but the worst-case running time is the same.

(i) **T** Given a weighted directed graph $G = (V, E, w)$ and a shortest path p from s to t, if we doubled the weight of every edge to produce $G'' = (V, E, w')$, then p is also a shortest path in G''.

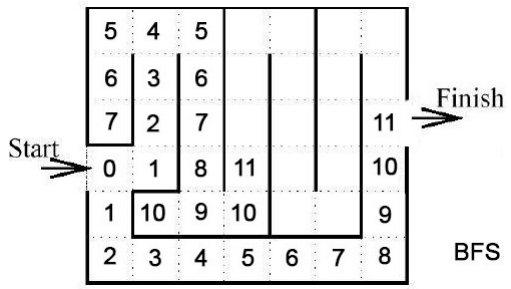
True. Multiplying edge weights by any positive constant factor pre-serves their relative order, as well as the relative order of any linear combination of the weights. All path weights are linear combinations of edge weights, so the relative order of path weights is preserved. This means that a shortest path in G will still be a shortest path in G''.

۲. [۳۶نمره] در هرکدام از شکل های زیر یک ماز وجود دارد و شما باید با الگوریتم خواسته شده، از نقطه ی شروع به نقطه ی پایان برسید. در صورت ملاقات کردن هرکدام از خانه های این ماز (که اطراف آن خط ممتد یا خط چین مشخص شده است) باید طول مسیر طی شده تا رسیدن به خانه ی فعلی را بنویسید.
تابع پتانسیل را در صورت نیاز تابع زیر در نظر بگیرید

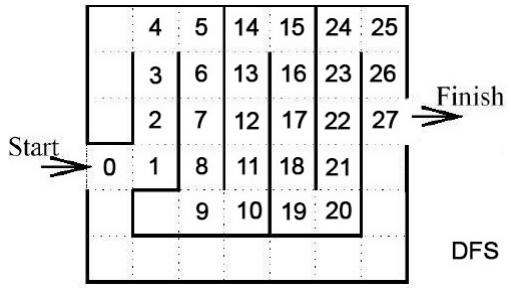
City Block Distance: $d = | x_1 - x_2 | + | y_1 - y_2 |$

اولویت ملاقات خانه ها در شرایط برابر به ترتیب: بالا، راست، پایین، چپ است.

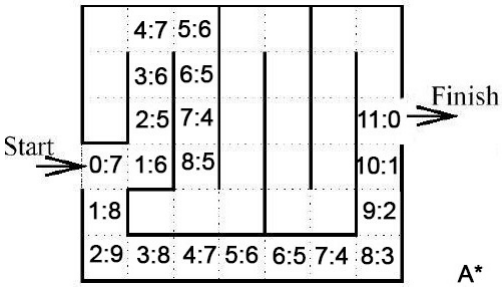
(a) BFS



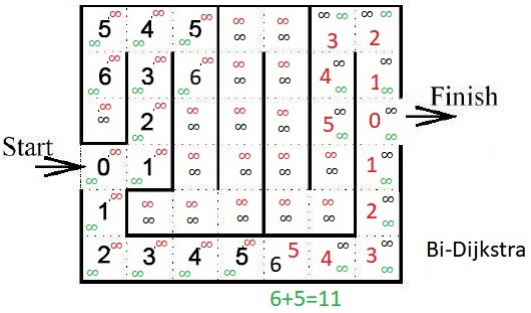
(b) DFS



(c) A*



(d) Bi-Directional Dijkstra



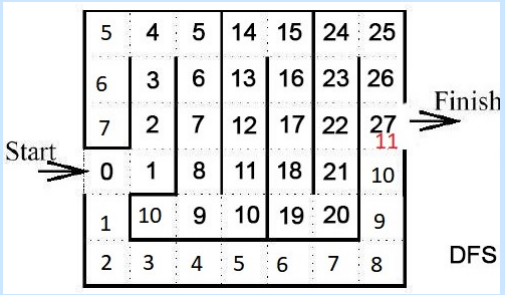
برای الگوریتم A^* عدد سمت چپ طول مسیر است و عدد سمت راست فاصله از مقصد است. محاسبه طول مسیر برای A^* همانطور که در اسلایدها موجود است و در کلاس هم توضیح داده شد طول مسیر به شکل زیر محاسبه میشود:

$l(u,v) = l(u,v) + \pi(v) - \pi(u)$

چون هنگام محاسبه مسیر از مبدا به خانه‌های مختلف جدول $\pi(s)$ در همه مسیرها ثابت است کوتاه‌ترین مسیر، مسیری است که کمترین $l(s,u) + \pi(u)$ را پیدا کند. در جدول زیر عدد سمت چپ $l(s,u)$ است و عدد سمت راست $\pi(u)$ میباشد.

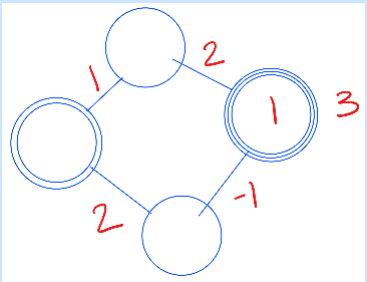
برای الگوریتم Dijkstra دوطرفه، اعداد سیاه فاصله از مبدا در گراف اصلی است. اعداد قرمز فاصله از مقصد در گراف واژگون است و اعداد سبز جمع فاصله از مبدا و مقصد است برای گره‌هایی که مقدار نهایی فاصله‌شان در گراف اصلی یا واژگون بدست آمده است (process شده‌اند).

اولین مسیری که پیمایش DFS از مبدا به مقصد پیدا میکند لزوما کوتاه‌ترین نیست. برای پیدا کردن کوتاه‌ترین مسیر لازم است کل گراف را پیمایش کند. به خاطر ابهام در پیدا کردن یک مسیر یا کوتاه‌ترین مسیر، هر دو جواب قابل قبول است.



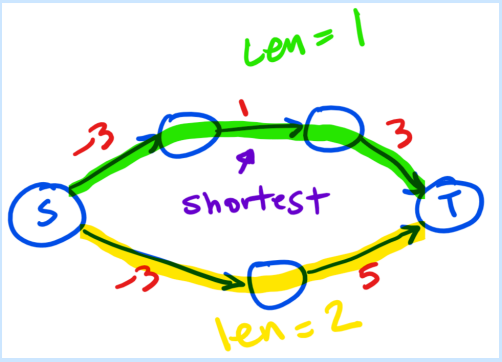
۳. [۱۷نمره] پیدا کردن کوتاه‌ترین مسیرها.

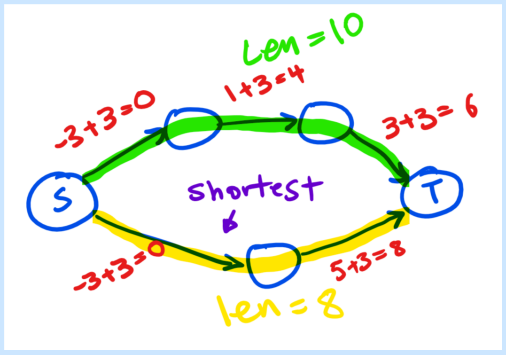
(آ) [۷نمره] گرافی با حداکثر چهار گره رسم کنید که الگوریتم Bellman-Ford جواب درست میدهد ولی الگوریتم Dijkstra جواب درست نمیدهد. گره‌ها را با یک دایره مشخص کنید و وزن یال‌ها را روی یال مشخص کنید. گره مبدا را با دو دایره تو در تو مشخص کنید و گرهی که الگوریتم Dijkstra برای آن کوتاه‌ترین فاصله را اشتباه محاسبه میکند را با سه دایره تو در تو مشخص کنید. فاصله‌ای که الگوریتم Dijkstra برای آن گره محاسبه میکند را بیرون دایره و فاصله‌ای که الگوریتم Bellman-Ford برای آن پیدا میکند را درون دایره بنویسید. هیچ توضیح اضافه‌ای لازم نیست.



(ب) [۱۰نمره] کوتاه‌ترین فاصله بین گره u و بقیه گره‌ها را در گراف $G(V, E, W)$ که دارای یال‌های با وزن منفی است ولی دارای دور منفی نیست محاسبه میکنیم. سپس برای حذف یال با وزن منفی به اندازه کوچکترین وزن، به وزن تمام یال‌ها اضافه میکنیم تا وزن تمام یال‌ها مثبت شود. الگوریتم Dijkstra را روی گراف تغییر یافته از گره u اجرا میکنیم. بدیهی است که مقدار کوتاه‌ترین فاصله در این دو گراف متفاوت خواهد بود. اما آیا کوتاهترین مسیر پیدا شده بین گره u و گره‌های دیگر در دو گراف یکسان است؟ اگر بله اثبات کنید. اگر خیر، مثال نقض بزنید.

خیر. مثال نقض:





۴. [۲۰نمره] سوال‌های زیر را در رابطه با متد MyAlgorithm جواب دهید.

(آ) خروجی این متد چیست؟

الگوریتم Prim برای محاسبه Minimum_Spanning_Tree است و وزن آن را برمیگرداند.

(ب) اگر PriorityQueue به صورت BinaryHeap پیاده سازی شده باشد پیچیدگی محاسباتی این متد چیست؟

Priority Queue Construction: $O(V)$
V times ExtractMin: $O(V \times \log V)$
Traversing Adjacency Matrix: $O(V^2)$
Updating costs vector: $O(E)$
E times ChangePriority: $O(E \times \log V)$
Total Complexity: $O(E \times \log V + V^2)$.
For dense graph E is $O(V^2)$. Therefore dense graph complexity is $O(V^2 \times \log V)$.

(ج) اگر PriorityQueue به صورت آرایه ساده پیاده‌سازی شده باشد پیچیدگی محاسباتی این متد چیست؟

Priority Queue Construction: $O(V)$
V times ExtractMin: $O(V^2)$
Traversing Adjacency Matrix: $O(V^2)$
Updating costs vector: $O(E)$
ChangePriority: $O(E \times 1)$
Total Complexity: $O(E + V^2)$.
For dense graph E is $O(V^2)$. Therefore dense graph complexity is $O(V^2)$

(د) اگر گره‌ها محل ساخت تعدادی خانه باشند و وزن یال‌ها فاصله اقلیدسی بین نقاط باشند، پیچیدگی محاسباتی کدام پیاده‌سازی بهتر است؟

وزن یال بین هر دو گره فاصله اقلیدسی آنها است. در نتیجه این گراف ژرف/dense است و در نتیجه پیچیدگی محاسباتی پیاده‌سازی PriorityQueue با آرایه مرتبه کمتری داشته و بهتر است.

```
۱   public int MyAlgorithm(int?[,] adjMatrix)
۲   {
۳       int nodeCount = adjMatrix.GetLength(0);
۴       bool[] processed = Enumerable.Repeat(false, nodeCount).ToArray();
۵       int[] costs = Enumerable.Repeat(int.MaxValue, nodeCount).ToArray();
۶       costs[0] = 0;
۷       var priorityQ = new PriorityQueue(
۸           keys: costs, values: Enumerable.Range(0,nodeCount));
۹       while (priorityQ.Count > 0)
۱۰      {
۱۱          var v = priorityQ.ExtractMinKey();
۱۲          for (int z = 0; z < nodeCount; z++)
۱۳              if (adjMatrix[v, z].HasValue)
۱۴                  if (!processed[z] && costs[z] > adjMatrix[v, z].Value)
۱۵                  {
۱۶                      costs[z] = adjMatrix[v, z].Value;
۱۷                      priorityQ.ChangePriority(z, costs[z]);
۱۸                  }
۱۹          processed[v] = true;
۲۰      }
۲۱      return costs.Sum();
۲۲  }
```

تشکر ویژه از آقای سهیل رستگار، خانم مریم سادات هاشمی و آقای امیر خاکپور برای همکاری در طراحی سوال‌های امتحان و کمک در برگزاری کوییز و پاسخ‌گویی به سوالات دانشجویان.

⋆ بخشی از سوالات این کوییز از درس طراحی و تحلیل الگوریتم دانشگاه MIT اقتباس شده است.

6.006: Introduction to Algorithms