গ্রাফ অ্যালগরিদম

শাফায়েত আশরাফ

দ্বিমিক প্রকাশনী

Scanned by CamScanner

কৃতজ্ঞতা স্বীকার

এটি আমার লেখা প্রথম বই। বইটি লেখার পেছনে কিছু মানুষের অবদান অনস্বীকার্য। মাহবুবুল হাসান শান্ত প্রতিটি অধ্যায়ের প্রায় প্রতিটি অক্ষর পড়ে ভুল ত্রুটি ধরিয়ে দিয়েছেন। তামিম শাহরিয়ার সুবিন নিয়মিত দৌড়ের উপর না রাখলে হয়তো বইটি লেখাই হত না। "প্রধান উপদেষ্টা" তানভীরুল ইসলাম অনবরত বিভিন্ন উপদেশ দিয়ে সাহায্য করেছেন। বইটি যখন শেষের দিকে তখন মোহাম্মদ সামিউল ইসলাম বেশ কিছু ভুল ত্রুটি ধরিয়ে দিয়েছেন।

আমার ব্লগের পাঠকের প্রতি কৃতজ্ঞ, তাঁরাই আমার লেখালেখির পেছনে অনুপ্রেরণা। এছাড়াও আমার বাবা-মা নিয়মিত খোঁজখবর নিয়েছেন বইয়ের অগ্রগতি নিয়ে।

বিশেষ কৃতজ্ঞতা আমার জীবনসঙ্গী লাজুকের প্রতি, তার উৎসাহেই বইটি লিখতে পেরেছি।

এবং আমি কৃতজ্ঞ প্রয়াত লেখক অভিজিৎ রায়ের প্রতি, আপনি না থাকলে আমার কোনোদিন লেখালেখি করা হতো না।

আমি জানি অনেক চেষ্টার পরেও এখনও বইয়ে অসংখ্য ভুল ত্রুটি থেকে গিয়েছে, সেজন্য আমি আন্তরিকভাবে দু:খিত।

হ্যাপি কোডিং!

শাফায়েত আশরাফ
www.shafaetsplanet.com/planetcoding

মানিক লাইব্রেরী
নীলক্ষেত, ঢাকা
০১৭৩৫৭৪২৯০৮

Scanned by CamScanner

Scanned by CamScanner

সূচিপত্র

ভূমিকা ১১
লেখক পরিচিতি ১৩
অধ্যায় ১ – গ্রাফ থিওরিতে হাতেখড়ি ১৫
১.১ – কনিসবার্গের সেতু ১৬
১.২ – অয়লারের আক্রমণ ১৭
১.৩ – অসাধ্যতার প্রমান ১৮
১.৪ – গ্রাফ কী? ১৯
অ্যাডজেসেন্ট নোড (Adjacent node) ২১
সাবগ্রাফ (Subgraph) ২১
ডিরেক্টেড (Directed) গ্রাফ এবং আনডিরেক্টেড (Undirected) গ্রাফ ২১
ওয়েটেড (Weighted) ও আনওয়েটেড (Unweighted) গ্রাফ ২১
পাথ (Path) ২২
ডিগ্রি (Degree) ২৩
ট্রি (Tree) ২৪
১.৫ – কিছু বিশেষ গ্রাফ ২৪
কমপ্লিট গ্রাফ (Complete Graph) ২৪
সাইকেল (Cycle) ২৪
হুইল গ্রাফ (Wheel Graph) ২৪
বাইপারটাইট গ্রাফ (Bipartite Graph) ২৪
অধ্যায় ২ – গ্রাফ উপস্থাপন ২৭
২.১ – অ্যাডজেসেন্সি ম্যাট্রিক্স (Adjacency Matrix) ২৭
অ্যাডজেসেন্সি ম্যাট্রিক্স ব্যবহার করার সমস্যা ২৮
অ্যাডজেসেন্সি ম্যাট্রিক্স ব্যবহার করার সুবিধা ২৯

Scanned by CamScanner

২.২ – অ্যাডজেসেন্সি লিস্ট (Adjacency List) ২৯
অ্যাডজেসেন্সি লিস্ট ব্যবহার করার সুবিধা ৩০
অ্যাডজেসেন্সি লিস্ট ব্যবহার করার অসুবিধা ৩০
অধ্যায় ৩ – ব্রেডথ ফার্স্ট সার্চ (Breadth First Search) ৩১
৩.১ – অ্যালগরিদমের বিবরণ ৩১
2D গ্রিডে বিএফএস ৩৭
৩.২ – বাইকালারিং (Bicoloring) ৩৭
অধ্যায় ৪ – ডায়াক্সট্রা অ্যালগরিদম (Dijkstra Algorithm) ৪০
৪.১ – পাথ রিলাক্সেশন ৪০
৪.২ – অ্যালগরিদমের বিবরণ ৪১
নেগেটিভ সাইকেল? ৪৪
২য় সেরা সংক্ষিপ্ততম পথ ৪৫
অধ্যায় ৫ – ফ্লয়েড ওয়ার্শল অ্যালগরিদম (Floyd Warshall Algorithm) ৪৬
৫.১ – অ্যালগরিদমের বিবরণ ৪৬
ট্রান্সিটিভ ক্লোজার (Transitive Closure): ৫১
অধ্যায় ৬ – ডিসজয়েন্ট সেট (Disjoint Set) ৫৩
অধ্যায় ৭ – মিনিমাম স্প্যানিং ট্রি (Minimum Spanning Tree) ৬১
৭.১ – প্রিম অ্যালগরিদম (Prim's Algorithm) ৬১
৭.২ – ক্রুসকাল অ্যালগরিদম (Kruskal's Algorithm) ৬৭
২য় সেরা মিনিমাম স্প্যানিং ট্রি ৭০
অধ্যায় ৮ – টপোলজিকাল সর্টিং (Topological Sorting) ৭১
অধ্যায় ৯ – বেলম্যান ফোর্ড অ্যালগরিদম (Bellman Ford Algorithm) ৭৭
অধ্যায় ১০ – স্টেবল ম্যারেজ (Stable Marriage) ৮৪
অধ্যায় ১১ – ডেপথ ফার্স্ট সার্চ (Depth First Search) ৮৭
অধ্যায় ১২ – আর্টিকুলেশন পয়েন্ট (Articulation Point) এবং ব্রিজ (Bridge) ৯৬

Scanned by CamScanner

১২.১ – আর্টিকুলেশন পয়েন্ট (Articulation Point) ৯৬
১২.২ – ব্রিজ (Bridge) ১০২
ইউনিক পাথ প্রবলেম ১০৩
অধ্যায় ১৩ – স্ট্রংলি কানেক্টেড কম্পোনেন্ট (Strongly Connected Component) ১০৪
অধ্যায় ১৪ – ম্যাক্সিমাম ফ্লো (Maximum Flow) ১০৯
১৪.১ – কয়েকটি ভ্যারিয়েশন ১১৫
#১ একাধিক সোর্স/সিংক থাকলে কী করবে? ১১৫
#২ প্রতিটি নোডের নির্দিষ্ট ক্যাপাসিটি থাকলে কী করবে? ১১৬
#৩ এজ ডিসজয়েন্ট পাথ কীভাবে বের করবে? ১১৭
১৪.২ – বাইপারটাইট ম্যাচিং ১১৮
১৪.৩ – মিন-কাট প্রবলেম ১২১
শেষ কথা ১২৩

Scanned by CamScanner

ভূমিকা

গ্রাফ অ্যালগরিদমে আমার দুর্বলতা গোপন কিছু নয়। তাই লেখকের কাছ থেকে যখন তার বইয়ের মুখবন্ধ লেখার প্রস্তাব পেলাম তখন খুশি আর আমার ধরে না। কারণ এরকম প্রস্তাব এই প্রথম, বিনামূল্যে গ্রাফ অ্যালগরিদম শিখে ফেলা যাবে। সর্বোপরি এই বই দেখিয়ে নিজেকে গ্রাফ বিশেষজ্ঞ প্রমাণ করা যাবে। আরেকটি কারণ হলো আমি এবং শাফায়েত প্রোগ্রামিং কনটেস্টের দুই ঘরানার লোক (DU-BUET অথবা Codemarshal-Hackerrank যেভাবেই দেখি না কেন)। কিন্তু সবকিছুর আগে আমরা বাংলাদেশের ভালো চাই এবং আমাদের কাজে সেটাই ফুটে ওঠা উচিত। সেই ফুটিয়ে উঠানোর কাজটা সবচেয়ে ভালোভাবে করা সম্ভব এই বই এর মুখবন্ধ লেখার মাধ্যমে। পুরো বইটি পড়ার সময় হয়নি এখনো, কিন্তু যতদুর দেখেছি তাতেই মনে করি বইটি বোঝা বেশ সহজ, অযথা গাণিতিক প্রতীক / চিহ্ন ইত্যাদি ব্যবহার করে লেখক পান্ডিত্ব দেখানোর চেষ্টা করেননি এবং অনেক বেশি টপিক নিয়ে আলোচনারও চেষ্টা করেননি।

আমাদের দেশের ছেলে মেয়েদের ইংরেজিতে দুর্বলতা অনেক বেশি, কিন্তু এই দুর্বলতা ভালো প্রোগ্রামার হওয়ার পথে কোনো অন্তরায় হওয়া উচিত নয়। যদি তাই হতো তাহলে রাশিয়া, চীন থেকে বিশ্বের সব ভালো ভালো প্রোগ্রামাররা বের হয়ে আসত না আর আইসিপিসিতে ইংরেজিভাষী দেশগুলোকে মেডেলের জন্য হা-হুতাশ করতে হতো না। কাজেই কম্পিউটার বিজ্ঞানের বিভিন্ন বিষয়ের ওপর আরও বাংলা বই বের হওয়া উচিত এবং সেদিক দিয়ে এই বইটি একটি ভালো সংযোজন। এ ধরনের বই পড়ে আমাদের দেশের ছেলেমেয়েরা অনেক কম বয়সেই খুব ভালো প্রোগ্রামার হয়ে উঠতে পারবে।

বিশ্ব মানের প্রোগ্রামার তৈরির জন্য প্রোগ্রামিং প্রতিযোগিতা একমাত্র উপায় না হলেও খুবই গুরুত্বপূর্ণ একটি প্রক্রিয়া। তবে আমাদের দেশে ভালো প্রোগ্রামার তৈরিতে এর গুরুত্ব বিশ্বের উন্নত দেশের চেয়ে অনেক বেশি। কারণ, আমাদের দেশের প্রচলিত শিক্ষা ব্যবস্থা এখনো যুক্তিনির্ভর চিন্তা ভাবনার জন্য অনুকুল নয়। সেদিক থেকেও এই বইটি এর গুরুত্ব অপরিসীম। এই বই এ আলোচিত গ্রাফ অ্যালগরিদম গুলো ভালোভাবে বুঝলে প্রোগ্রামিং প্রতিযোগিতার অন্তত ৮০ ভাগ গ্রাফ সমস্যা সমাধান করা সম্ভব।

প্রোগ্রামিং শেখার জন্য বাংলায় বেশ কিছু বই রয়েছে। কিন্তু মূলত প্রোগ্রামিংয়ের একদম প্রারম্ভিক বিষয়গুলো নিয়ে সেখানে আলোচনা করা হয়েছে। অ্যালগরিদম শেখার জন্য ভালো বাংলা বই এখনো সেভাবে ছাপা হয়ে বের হয়নি। পিডিএফ হিসাবে হয়ত বেশ কিছু বই পাওয়া যাচ্ছে। সেদিক থেকেও এই বইটিকে একটি মাইলফলক হিসাবে দেখা যায়। সব মিলিয়ে আমি লেখককে ধন্যবাদ দিতে চাই এরকম একটি বই লেখার জন্য। আমি আশা করি সময়ের সঙ্গে সঙ্গে বইটি

Scanned by CamScanner

আরও উন্নত হবে এবং সঙ্গে সঙ্গে কম্পিউটার ইঞ্জিনিয়ারিংয়ে আরও অনেক ভালো ভালো বাংলা বই প্রকাশিত হবে। তখন আমরা জাপানিদের মতো বিশ্ববিদ্যালয় পর্যায়েও মাতৃভাষায় শিক্ষা দিতে পারব। ৫২ এর ভাষা আন্দোলন সেদিনই পরিপূর্ণভাবে সার্থক হবে।

শাহরিয়ার মঞ্জুর,

সভাপতি, বাংলাদেশ অ্যাসোসিয়েশন অব প্রবলেমসেটারস
জাজিং ডিরেক্টর, এসিএম আইসিপিসি ঢাকা রিজিয়নাল (২০০৪-২০১৫)
বিচারক, এসিএম আইসিপিসি ওয়ার্ল্ড ফাইনাল (২০০৩-২০১৬)
জাজিং ডিরেক্টর, এসিএম আইসিপিসি ঢাকা রিজিয়নাল (২০০৩-২০১৬)
এছাড়াও ২০০০ সাল থেকে ভ্যালাদলিদ অনলাইন জাজ (Valladolid Online Judge) এর সঙ্গে জড়িত

Scanned by CamScanner

লেখক পরিচিতি

শাফায়েত আশরাফের জন্ম ১৯৯০ সালে। তিনি আদমজী ক্যান্টনমেন্ট পাবলিক স্কুল থেকে মাধ্যমিক ও রাজউক উত্তরা মডেল কলেজ থেকে উচ্চ মাধ্যমিক পরীক্ষা সম্পন্ন করেন। এরপর তিনি ঢাকা বিশ্ববিদ্যালয়ের কম্পিউটার বিজ্ঞান বিভাগে ভর্তি হন। বিশ্ববিদ্যালয়ে পড়াকালীন সময়ে তিনি নিয়মিত প্রোগ্রামিং প্রতিযোগিতায় অংশগ্রহণ করতেন। ২০১২ সালে তার দল জাতীয় পর্যায়ের প্রোগ্রামিং প্রতিযোগিতায় বিজয়ী হয়।

শাফায়েতের বাংলা ব্লগে অ্যালগরিদম নিয়ে ৫০টিরও অধিক প্রবন্ধ আছে (www.shafaetsplanet.com/blog)। তিনি বাংলাদেশ ইনফরমেটিক্স অলিম্পিয়াড এবং ন্যাশনাল হাই স্কুল প্রোগ্রামিং প্রতিযোগিতার বিচারক এবং প্রবলেমসেটার হিসাবে কাজ করেছেন। এছাড়াও তিনি এনসিপিসি (National Collegiate Programming Contest) সহ বিভিন্ন জাতীয় পর্যায়ের প্রোগ্রামিং প্রতিযোগিতায় বিচারকের দায়িত্ব পালন করেছেন।

ঢাকা বিশ্ববিদ্যালয় থেকে অনার্স ডিগ্রী লাভের পর তিনি ন্যাশনাল ইউনিভার্সিটি অফ সিঙ্গাপুর (National University of Singapore) এর সেন্টার ফর কোয়ান্টাম টেকনোলজিতে ইন্টার্ন রিসার্চার হিসাবে কাজ করছেন। বর্তমানে তিনি হ্যাকাররর‍্যাঙ্ক ডট কম (hackerrank.com) এ সফটওয়্যার ইঞ্জিনিয়ার এবং প্রবলেম কিউরেটর হিসাবে কাজ করছেন।

Scanned by CamScanner

অধ্যায় ১ – গ্রাফ থিওরিতে হাতেখড়ি

কেউ যখন ছবি আঁকে বা গান বাঁধে তখন তাকে আমরা বলি সৃষ্টিশীল। কারণ সে এমন কিছু সৃষ্টি করছে যা আগে ছিল না। সেই হিসাবে একজন গণিতবিদের সৃষ্টিশীলতা কোথায় তা চট করে ধরা যায় না। আসলেই তো, ক্যালকুলাসের ডিফারেন্সিয়েশন বা ইন্টিগ্রেশন, সেসব তো সুনির্দিষ্ট নিয়ম মেনে ধাপে ধাপে এগিয়ে গেলেই সমাধান হয়। ছোট ক্লাসের সুদের হার বা লাভ-ক্ষতি বের করার অঙ্ক তাতেই বা সৃষ্টিশীলতা কোথায়? কে সেরা গণিতবিদ সেটা বুঝতে তাই কে বেশি বেশি সমস্যা সমাধান করতে পারছে তার চেয়ে বেশি গুরুত্ব দেওয়া হয় কার সমাধান প্রক্রিয়ার অভিনবত্ব বেশি। কেউ হয়তো গৎবাঁধা নিয়মে না এগিয়ে জানা সূত্রগুলোকে ভিন্ন কোনো উপায়ে প্রয়োগ করেছে। কেউ হয়তো সমাধানের জন্য প্রথমে নিজেই কোনো নতুন সূত্র বানিয়ে নিয়েছে। আবার এমনও অনেক নজির আছে যে কোনো সমস্যা সমাধান করতে গিয়ে গণিতের আস্ত একটা শাখাই খুলে বসেছে কেউ কেউ। এই শেষ প্রকারের গণিতবিদের নামের আগে আমরা "মহান গণিতবিদ" বা "মহান গণিতবেত্তা" টাইটেল জুড়ে দেই। গ্রাফ থিওরি নামক গাণিতিক তত্ত্বের সূচনা হয়েছে তেমন এক মহান গণিতবিদ লিওনার্দ অয়লারের হাত ধরে। একই ভাবে ক্যালকুলাস অফ ভেরিয়েন্সের সূচনা হয়েছে নিউটনের দ্বারা, গ্রুপ থিওরির সূচনা করেছেন গ্যালোয়া ইত্যাদি...।

কেউ হয়তো ভাবতে পারে, নিশ্চয়ই খুব কঠিন, একেবারে অবোধগম্য কোনো গাণিতিক সমস্যার সমাধান করতে গিয়ে বুঝি এই গণিতবিদরা গণিতের নতুন শাখাই খুলে ফেলেন। আসলে তা নয়, বেশিরভাগ সময়ই দেখা যায় সমস্যাগুলো খুবই সোজাসাপ্টা এবং বাহ্যিকভাবে একেবারেই মামুলি। যেমন, কনিগসবার্গ শহরের সাতটি সেতু ঘুরে ঘুরে দেখা সংক্রান্ত একটা ধাঁধা সমাধান করতে গিয়ে অয়লার গ্রাফ থিওরির গোড়াপত্তন করেন, ক্যালকুলাস অফ ভেরিয়েন্সের গোড়াপত্তন করেন নিউটন একটি হেলানো বক্রতলে গড়িয়ে পড়া গোলকের গতি সংক্রান্ত সমস্যার সমাধান করতে গিয়ে, আর গ্যালোয়া গ্রুপ থিওরির গোড়াপত্তন করেন সাদামাটা দেখতে বহুপদী সমীকরণের সমাধান নির্ণয় করতে গিয়ে। খেলাচ্ছলে কোনো সমস্যার সমাধান করতে গিয়ে জ্ঞানের একটা শাখা সৃষ্টি করে ফেলার উদাহরণ গণিত আর বিজ্ঞানে ভূরি ভূরি। এ ধরনের কাজ যারা করেন তারাই হচ্ছেন এসব ক্ষেত্রের প্রথম সারির সৃষ্টিশীল ব্যক্তি। তাদের সব আবিষ্কারের গল্পগুলোও অভিনব। কিন্তু আমাদের এই লেখা যেহেতু গ্রাফ থিওরি নিয়ে সেহেতু আমরা শুধু অয়লার কিভাবে ধাঁধা সমাধান করতে গিয়ে গ্রাফ থিওরি বানিয়ে ফেললেন সেটি দেখব।

Scanned by CamScanner

১.১ – কনিসবার্গের সেতু

কনিগসবার্গ (Königsberg) শহরের মধ্যে দিয়ে বয়ে গেছে একটি নদী। নাম প্রেগেল। নদীটি শুধু এঁকেবেঁকে বের হয়েই যায়নি, বরং প্রকৃতির অদ্ভুত খেয়ালে নগরীর মধ্যে একটি দ্বীপের মত সৃষ্টি করেছে। আর বাকি অংশকে ভাগ করেছে তিন ভাগে (চিত্র ১.১)।

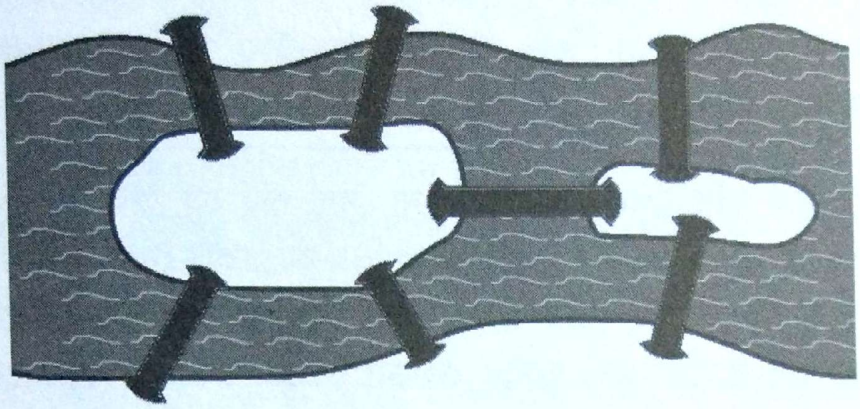
চিত্র ১.১: কনিসবার্গ শহরের অংশবিশেষের মানচিত্র

বাসিন্দারা যেন ঠিকঠাকমতো চলাফেরা করতে পারে সে জন্য এইসব অংশকে যুক্ত করতে 7টি সেতু নির্মান করেছে নগর কর্তৃপক্ষ। আর ম্যাপে সেই চিত্র দেখতে গিয়েই কি না, কেউ একজন হঠাৎ প্রশ্ন করে বসেছে, নগরীর রাস্তা দিয়ে কি এমনভাবে ঘোরাঘুরি করা সম্ভব যেন 7টি সেতুই শুধুমাত্র একবার করে অতিক্রম করা হয়? অর্থাৎ একবার যদি কোনো সেতু পার হয়ে এক অংশ থেকে অন্য অংশে যাওয়া হয়, ওই সেতু আর ব্যবহার করা যাবে না। তারপর কেটে গেছে অনেক বছর, কাগজে সেতু আর নগরের অংশগুলোর চিত্র এঁকে এঁকে অনেকেই চেষ্টা করেছে এমন একটি পথ বের করতে যেন মূল শর্তগুলো মেনে 7টি সেতুই অতিক্রম করা যায়। সবাই ব্যর্থ। কেউ হয়তো বলে, না এমন সম্ভবই নয়। কিন্তু কাজটি যে অসম্ভব সেটিই বা নিশ্চিত হওয়া যায় কীভাবে? কে জানে, খুব বুদ্ধিমান কেউ এসে হয়তো কোনো উপায় বের করে ফেলবে! কোনো সমস্যা সমাধান করতে না পারলেই তো আর ঘোষণা দেওয়া যায় না সমস্যাটি সমাধানের অযোগ্য। বরং যদি একেবারে যুক্তি দিয়ে প্রমাণ করা যায় যে কাজটি অসাধ্য তখনই শুধু ওই না-বোধক ঘোষণা চলে। আর সেই কাজটিই প্রথম করেছেন লিওনার্দ অয়লার। বোঝাই যাচ্ছে, মূল সমস্যাটি ছোটদের কাগজে ছাপানোর মত একটি ধাঁধা কিন্তু তা নিয়ে মাথা ঘামাতে গিয়ে লিওনার্দ অয়লার কী অভাবনীয় কাজটিই না করে ফেললেন সেটিই আমরা দেখব।

Scanned by CamScanner

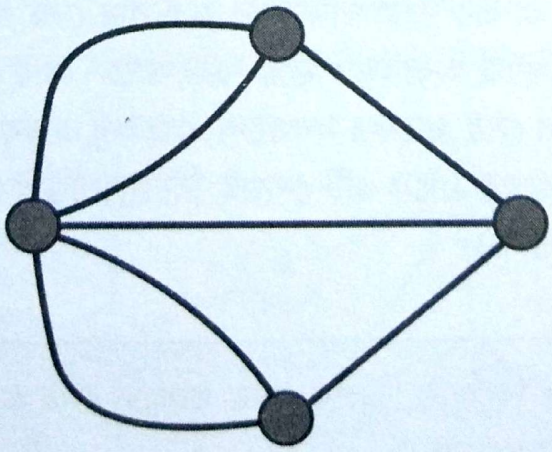
১.২ – অয়লারের আক্রমণ

অয়লার প্রথমেই ম্যাপ থেকে ভবন, তোরণ, রাস্তাঘাট এই সবকিছুকে বাদ দিয়ে মূল সমস্যার জন্য প্রয়োজনীয় অংশটিকে আলাদা করে ফেললেন চিত্র ১.২ এর মতো করে। তারপর তিনি খেয়াল করলেন, নগরীর ভাগগুলোর কোনো একটিতে কীভাবে কেউ হাটছে তা কোনো ব্যাপার নয়, স্রেফ সেখানে পৌঁছেছে কি না সেটিই বিবেচ্য।



চিত্র ১.২: কনিসবার্গ শহরের মানচিত্রে সরলীকৃত চিত্র

সেই সঙ্গে গুরুত্বপূর্ণ হচ্ছে সেতুগুলো, যেগুলো নগরীর বিভিন্ন অংশকে যুক্ত করেছে। তাই ১.২ নং চিত্রকে তিনি আরও সরলীকরণ করে চিত্র ১.৩ এর মতো করে আঁকলেন। এখন আর এখানে বিভ্রান্তি সৃষ্টি করার মতো বাড়তি কিছু নেই। পুরো সমস্যাটিকে ছোট ছোট বৃত্ত আর তাদের যুক্তকারী দাগে বদলে ফেলা হয়েছে। এই চিত্রটিকেই আমরা বলি **গ্রাফ (Graph)**। গ্রাফের এই ছোট ছোট বৃত্তগুলোকে বলি **নোড (Node)** আর তাদের যুক্তকারী রেখাগুলোকে বলি **এজ (Edge)**।



চিত্র ১.৩: কনিসবার্গ শহরের মানচিত্র থেকে তৈরী গ্রাফ

Scanned by CamScanner

কোনো নোডে কতগুলো এজ এসে যুক্ত হয়েছে তাকে বলি ওই নোডের ডিগ্রি (Degree)। এই যে একটি সমস্যার অপরিহার্য অংশকে স্রেফ কিছু নোড আর এজের মাধ্যমে এঁকে ফেলা এই পদ্ধতিটিই অয়লারের অমর কীর্তিগুলোর একটি। অবশ্য শুধু এঁকে থেমে গেলে ব্যাপারটি কোনো কাজের হতো না। তিনি এ ধরনের গ্রাফকে যুক্তি দিয়ে আর গাণিতিক সূত্র দিয়ে বিশ্লেষণের নানান উপায়ও বের করেছেন। এখন আমরা দেখব অয়লার কীভাবে তার গ্রাফ থেকে যুক্তি দিয়ে প্রমাণ করলেন যে কনিগসবার্গের সেতু সংক্রান্ত সমস্যাটির কোনো সমাধান নেই।

১.৩ – অসাধ্যতার প্রমান

চিত্র ১.৩ থেকে খেয়াল করি, যখন আমরা কোনো সেতু পার হই, অর্থাৎ গ্রাফটির কোনো এজ ব্যবহার করে ফেলি, তখন আমরা এক নোড থেকে ভিন্ন আরেকটি নোডে উপনীত হই। আর এই নোড পরিবর্তনে ব্যবহৃত এজটি পুনরায় ব্যবহার করা নিষিদ্ধ। ফলে ওখান থেকে বের হতে ভিন্ন আরেকটি এজ ব্যবহার করতে হয়। অর্থাৎ কোনো নোডে বাইরে থেকে ঢুকে বেরিয়ে গেলে নোডটির দুটি করে সংশ্লিষ্ট এজ ব্যবহৃত হয়ে যায়। ফলে কোনো নোডের সংশ্লিষ্ট এজ যদি বিজোড় সংখ্যক হয় (অর্থাৎ নোডটির ডিগ্রি বিজোড়) এবং আমরা যদি গ্রাফের ভিন্ন কোনো নোড থেকে যাত্রা শুরু করি, তাহলে পুরো গ্রাফটি যেভাবেই ঘুরি না কেন এই বিজোড় ডিগ্রির নোডটিতে আমরা ফিরে আসতে আসতে এমন অবস্থায় পৌঁছাব যখন আর বের হওয়ার উপায় থাকবে না। যদি আমাদের শুরুর নোডেরও ডিগ্রি বিজোড় হয়, আর শেষ নোডের ডিগ্রি বিজোড় হয় আর বাকি সব নোডের ডিগ্রি হয় জোড় তাহলে হয়তো চালাক কেউ এমন একটা পথ বের করতে পারে যেটি শুরু হয় একটি বিজোড় ডিগ্রির নোড থেকে, তারপর সবগুলো এজ ঘুরে শেষ হয় অন্য বিজোড় ডিগ্রির নোডে। কিন্তু গ্রাফে যদি দুই এর অধিক নোড থাকে যাদের ডিগ্রি বিজোড়, তখন যত চালাক পরিব্রাজকই হোক না কেন, এমন বিজোড় নোডে গিয়ে সে পৌঁছাবে যেখান থেকে বের হওয়ার কোনো উপায় নেই, আবার যাত্রা থামানোর ও উপায় নেই, কারণ তৃতীয় কোনো বিজোড় ডিগ্রির নোডের একটি এজ নিশ্চই অব্যবহৃত হয়ে পড়ে আছে। তাই কোনো গ্রাফে দুই এর অধিক বিজোড় ডিগ্রির নোড থাকলে সেই গ্রাফের সবগুলো এজ শুধু একবার করে পরিভ্রমণ করা সম্ভব নয়। যেহেতু কনিগসবার্গ শহরের গ্রাফে 4টি নোডই বিজোড় ডিগ্রির সেহেতু এই সেতু সমস্যার কোনো সমাধান থাকতে পারে না।

কোনো গ্রাফে দুই এর অধিক বিজোড় ডিগ্রির নোড থাকলে সেই গ্রাফের সবগুলো এজ শুধুমাত্র একবার করে পরিভ্রমণ করা সম্ভব নয়।

Scanned by CamScanner

গ্রাফ অ্যালগরিদমের অনেক ব্যবহার বাস্তবে আছে। যেমন:

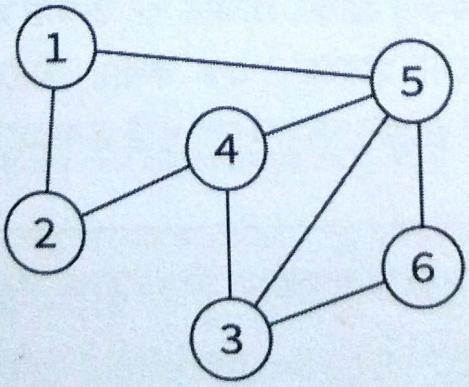
- একটি শহর থেকে আরেক শহরে অথবা এক এয়ারপোর্ট থেকে আরেক এয়ারপোর্টে যাওয়ার সংক্ষিপ্ততম পথ তুমি গ্রাফ অ্যালগরিদম দিয়ে বের করতে পার। সার্ভার থেকে একটি ওয়েবপেইজ তোমার পিসিতে পৌঁছাতে অনেকগুলো পয়েন্ট পার করে আসতে হয়, গ্রাফ থিওরি দিয়ে এক পয়েন্ট থেকে আরেকটিতে যাওয়ার পথ খুঁজে বের করা হয়। গুগল ম্যাপ তোমাকে পথ দেখানোর জন্য বেশ কিছু গ্রাফ অ্যালগরিদম ব্যবহার করে।
- তুমি শহরে নতুন কিছু ফ্লাইওভার বানাতে চাও। কোথায় ফ্লাইওভার বানালে বেশি কার্যকর হবে, কোন অংশে ট্রাফিক জ্যামের ওপর কীভাবে প্রভাব পড়বে এসবই গ্রাফ থিওরি ব্যবহার করে তোমাকে জেনে নিতে হবে।
- দুটি রাসায়নিক যৌগের আণবিক গঠন ভিন্ন নাকি সেটি নির্ণয় করা যায় গ্রাফ দিয়ে মডেলিং করে।

এরকম আরও অনেক ব্যবহার আমরা দেখব গ্রাফ অ্যালগরিদমের বিভিন্ন অ্যাপ্লিকেশন দেখার সময়।

অ্যালগরিদম শেখার আগে আমরা গ্রাফ সম্পর্কিত কয়েকটি সংজ্ঞা দেখব। শুধুমাত্র যেসব সংজ্ঞা শুরুতে জানা দরকার সেগুলো এখানে আছে, বাকিগুলো আমরা অ্যালগরিদম শেখার সময় প্রয়োজন মত শিখে নিব।

১.৪ – গ্রাফ কী?

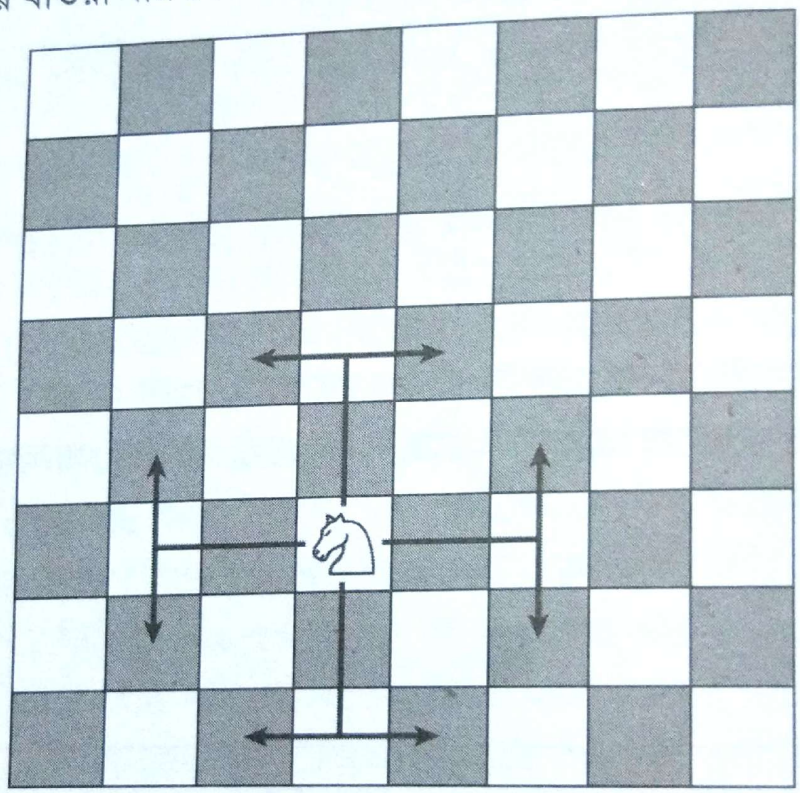
ধরা যাক 6টি শহরকে আমরা 1, 2, 3, 4, 5, 6 দিয়ে চিহ্নিত করলাম। এবার যে শহর থেকে যে শহরে সরাসরি রাস্তা আছে তাদের মধ্যে লাইন টেনে দিলাম (চিত্র ১.৪)।



চিত্র ১.৪: ছয়টি শহরের মানচিত্রের গ্রাফ

Scanned by CamScanner

এটি খুবই সাধারণ একটি গ্রাফ যেখানে কিছু শহরের মধ্যের রাস্তাগুলো দেখানো হয়েছে। গ্রাফ থিওরির ভাষায় শহরগুলোকে বলা হয় নোড (Node) বা ভারটেক্স (Vertex) আর রাস্তাগুলোকে বলা হয় এজ (Edge)। গ্রাফ হলো কিছু নোড আর কিছু এজের একটি কালেকশন বা সংগ্রহ। নোড হলো একটি অবজেক্ট যেটি দিয়ে গ্রাফে অনেক কিছু বোঝাতে পারে, কোনো গ্রাফে হয়তো নোড দিয়ে শহর বোঝায়, কোনো গ্রাফে এয়ারপোর্ট, কোনো গ্রাফে আবার হয়তো দাবার বোর্ডের একটি ঘর বোঝাতে পারে! আর এজ দিয়ে বোঝায় নোডগুলোর মধ্যের সম্পর্ক। কোনো গ্রাফে এজ দিয়ে দুটি শহরের দূরত্ব বোঝাতে পারে, কোনো গ্রাফে এক এয়ারপোর্ট থেকে আরেক এয়ারপোর্টে যাওয়ার সময় বোঝাতে পারে, আবার দাবার বোর্ডে একটা ঘরে ঘোড়া থাকলে সেই ঘর থেকে কোন ঘরে যাওয়া যায় সেটাও বোঝাতে পারে।



চিত্র ১.৫: দাবা বোর্ডে ঘোড়ার সম্ভাব্য চালের গ্রাফ

চিত্র ১.৫ এর দাবার বোর্ডটিও একটি গ্রাফ। প্রতিটি ঘর একটি করে নোড। যে ঘরে ঘোড়া আছে সেখান থেকে এজগুলো দেখানো হয়েছে। এক কথায় নোডের কাজ কোনো এক ধরনের অবজেক্টকে রিপ্রেজেন্ট করা আর এজের কাজ হলো দুটি অবজেক্টের মধ্যে সম্পর্কটি দেখানো।

নোডের কাজ কোনো এক ধরনের অবজেক্টকে রিপ্রেজেন্ট করা আর এজের কাজ হলো দুটি অবজেক্টের মধ্যে সম্পর্কটি দেখানো।

Scanned by CamScanner

অ্যাডজেসেন্ট নোড (Adjacent node)

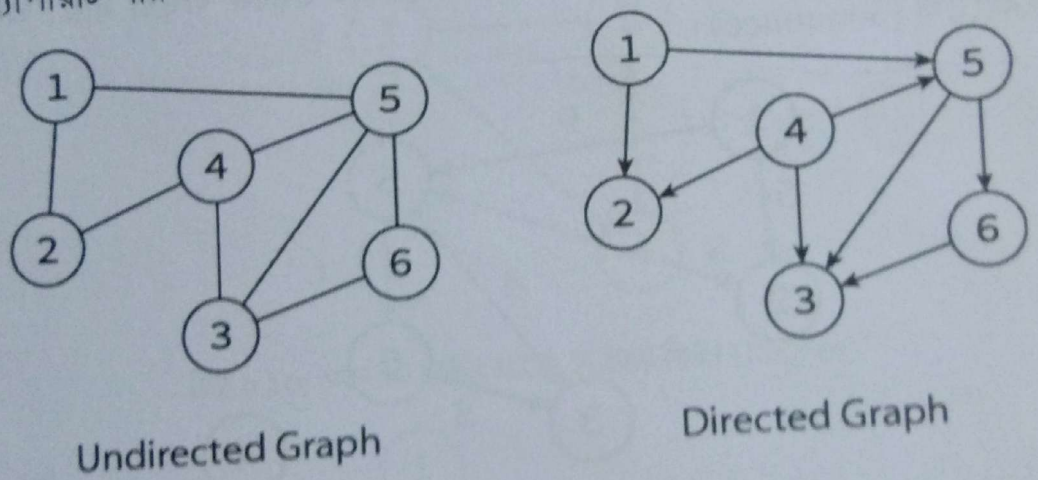
A নোড থেকে B নোডে একটি এজ থাকলে B নোডকে বলা হয় A নোডের অ্যাডজেসেন্ট নোড। সোজা কথায় অ্যাডজেসেন্ট নোড হলো এজ দিয়ে সরাসরি সংযুক্ত বা কানেক্টেড (Connected) নোড। একটি নোডের অনেকগুলো অ্যাডজেসেন্ট নোড থাকতে পারে।

সাবগ্রাফ (Subgraph)

সাবগ্রাফ হলো একটি গ্রাফের কিছু নোড এবং এজের সাবসেট (Subset)।

ডিরেক্টেড (Directed) গ্রাফ এবং আনডিরেক্টেড (Undirected) গ্রাফ

ডিরেক্টেড গ্রাফে এজগুলোতে তীরচিহ্ন থাকে, তার মানে এজগুলো হলো একমুখি (Unidirectional)। আনডিরেক্টেড গ্রাফে এজগুলো দ্বিমুখী (Bidirectional)। চিত্র ১.৬ দেখলেই ব্যাপারটি পরিষ্কার হবে।



চিত্র ১.৬: আনডিরেক্টেড (Undirected) ও ডিরেক্টেড (Directed) গ্রাফ

ওয়েটেড (Weighted) ও আনওয়েটেড (Unweighted) গ্রাফ

অনেক সময় গ্রাফে এজগুলোর পাশে ছোট করে ওয়েট (Weight) বা কস্ট (Cost) লেখা থাকতে পারে। এই ওয়েট দিয়ে অনেক কিছু বোঝাতে পারে, যেমন দুটি শহরের দূরত্ব কত কিলোমিটার অথবা রাস্তাটি দিয়ে যেতে কত সময় লাগে অথবা রাস্তা দিয়ে কয়টি গাড়ি এক সঙ্গে যেতে পারে ইত্যাদি। চিত্র ১.৭ এ একটি ডিরেক্টেড ওয়েটেড গ্রাফ দেখানো হয়েছে।

Scanned by CamScanner

ট্রি (Tree)

ট্রি হলো এমন একটি গ্রাফ যেখানে যেকোনো দুটি নোডের মধ্যে ঠিক 1টি মাত্র পাথ আছে। ট্রি গ্রাফে nটি নোড এবং $(n-1)$টি এজ থাকে।

একাধিক ট্রি দিয়ে তৈরি গ্রাফকে বলা হয় ফরেস্ট (Forest)। চিত্র ১.১১ তে একটি ট্রি দেখানো হয়েছে।

১.৫ – কিছু বিশেষ গ্রাফ

কমপ্লিট গ্রাফ (Complete Graph)

কমপ্লিট গ্রাফে প্রতিটি নোড প্রতিটি নোডের সঙ্গে সরাসরি এজ দিয়ে যুক্ত থাকে। nটি নোড নিয়ে গঠিত কমপ্লিট গ্রাফকে K_n দিয়ে প্রকাশ করা হয়। চিত্র ১.১২ এর গ্রাফটি K_4।

সাইকেল (Cycle)

nটি নোডের একটি সাইকেলকে C_n দিয়ে প্রকাশ করা হয়। প্রতিটি নোড $i < n$ তার পরবর্তী নোড $i+1$ এর সঙ্গে যুক্ত থাকে এবং শেষ নোডটি প্রথম নোডের সঙ্গে যুক্ত থাকে। চিত্র ১.১৩ তে চারটি নোডের একটি সাইকেল গ্রাফ C_4 দেখানো হয়েছে।

হুইল গ্রাফ (Wheel Graph)

সাইকেল গ্রাফে একটি অতিরিক্ত নোড যোগ করে অন্য সবগুলো নোডের সঙ্গে যোগ করে দিলে সেটিকে হইল গ্রাফ বলে যা দেখতে গাড়ির চাকার মতো। W_n দিয়ে হুইল গ্রাফ প্রকাশ করা হয়। চিত্র ১.১৪ তে W_4 দেখানো হয়েছে।

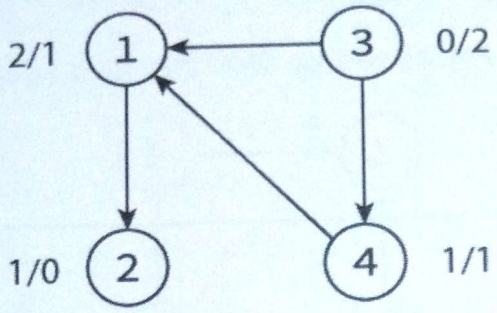
বাইপারটাইট গ্রাফ (Bipartite Graph)

বাইপারটাইট গ্রাফের নোডগুলোকে দুটি সেটে ভাগ করা যায়। একই সেটের দুটি নোডের মধ্যে কোনো এজ থাকে না।

Scanned by CamScanner

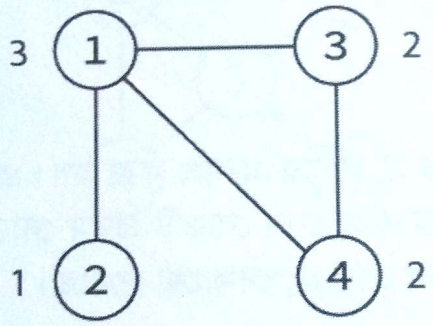
ডিগ্রি (Degree)

ডিরেক্টেড গ্রাফে একটি নোডে কয়টি এজ প্রবেশ করেছে তাকে ইনডিগ্রি (Indegree), আর কোনো নোড থেকে কয়টি এজ বের হয়েছে তাকে আউটডিগ্রি (Outdegree) বলে। চিত্র ১.৯ এ প্রতিটি নোডের ইনডিগ্রি আর আউটডিগ্রি দেখানো হয়েছে।

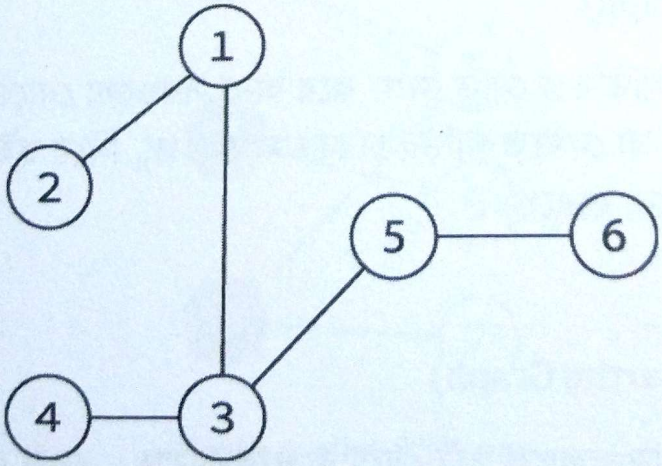


চিত্র ১.৯: ইনডিগ্রি (Indegree) / আউটডিগ্রি (Outdegree)

আনডিরেক্টেড গ্রাফে ইনডিগ্রি বা আউটডিগ্রি আলাদা করা হয়না। একটি নোডের যতগুলো অ্যাডজেসেন্ট নোড আছে সেই সংখ্যাটিই নোডটির ডিগ্রি।

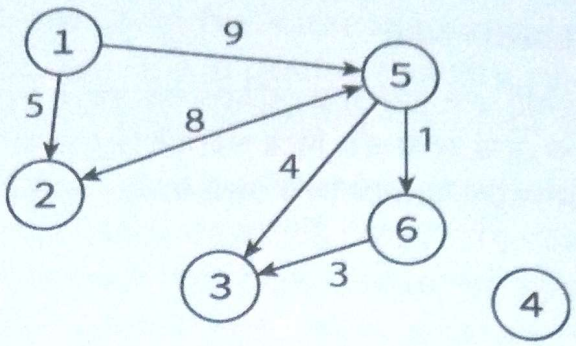


চিত্র ১.১০: আনডিরেক্টেড গ্রাফে নোডের ডিগ্রি (Degree)



চিত্র ১.১১: ট্রি (Tree)

Scanned by CamScanner

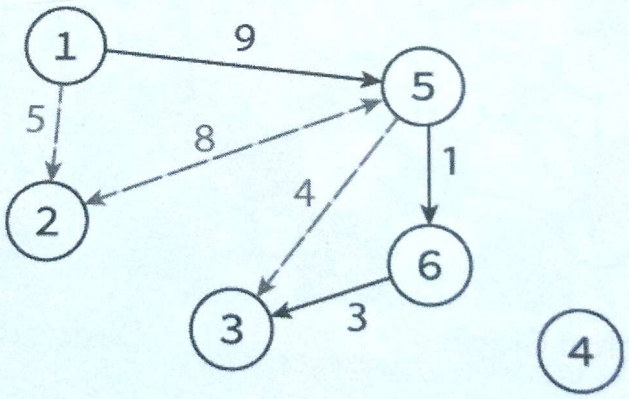


চিত্র ১.৭: ডিরেক্টেড ওয়েটেড গ্রাফ (Directed Weighted Graph)

আগের গ্রাফগুলো ছিলো আনওয়েটেড, সেক্ষেত্রে আমরা ধরে নেই সবগুলো এজের ওয়েটের মান 1। সবগুলো ওয়েট 1 হলে আলাদা করে লেখা দরকার হয়না।

পাথ (Path)

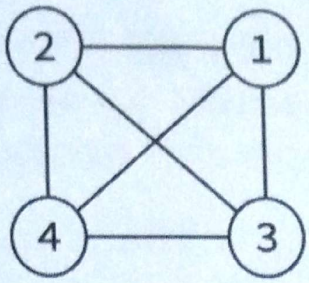
পাথ হলো যে এজগুলো ধরে একটি নোড থেকে আরেকটি নোডে যাওয়া যায়। অর্থাৎ এজের একটি সিকোয়েন্স (Sequence)।



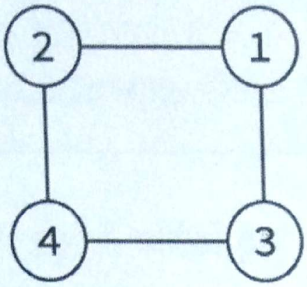
চিত্র ১.৮: নোড 1 → নোড 5 → নোড 6 → নোড 3 একটি পাথ

এক নোড থেকে আরেক নোডে যাওয়ার অনেকগুলো পাথ থাকতে পারে। যে পাথের কস্ট সব থেকে কম সেটিকে শর্টেস্ট পাথ (Shortest Path) বলে। চিত্র ১.৮ এ নোড 1 থেকে নোড 3 তে যাওয়ার একটি পাথ দেখানো হয়েছে (নোড 1 → নোড 5 → নোড 6 → নোড 3)।

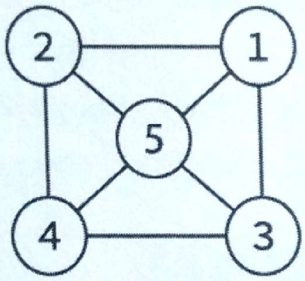
Scanned by CamScanner



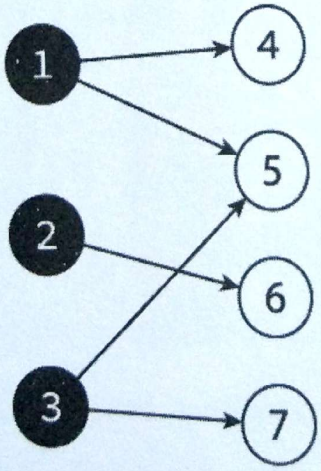
চিত্র ১.১২: চার নোড বিশিষ্ট একটি কমপ্লিট গ্রাফ (K_4)



চিত্র ১.১৩: চার নোড বিশিষ্ট একটি সাইকেল গ্রাফ (C_4)



চিত্র ১.১৪: হুইল গ্রাফ (W_4)



চিত্র ১.১৫: বাইপারটাইট গ্রাফ

Scanned by CamScanner

চিত্র ১.১৫ তে একটি বাইপারটাইট গ্রাফ দেখানো হয়েছে। দুটি সেটকে দুটি আলাদা রঙ দেওয়া হয়েছে। একটি গ্রাফকে যদি দুটি রঙ ব্যবহার করে এমনভাবে রঙ করা যায় যেন অ্যাডজেসেন্ট দুটি নোডের রঙ এক না হয়, তাহলে সেই গ্রাফটি বাইপারটাইট।

লক্ষ কর

একটি গ্রাফকে যদি দুটি রঙ ব্যবহার করে এমনভাবে রঙ করা যায় যেন অ্যাডজেসেন্ট দুটি নোডের রঙ এক না হয়, তাহলে সেই গ্রাফটি বাইপারটাইট।

এই অধ্যায়ে তুমি একদম প্রাথমিক কিছু জিনিস শিখেছ, পরের অধ্যায়ে আমরা শিখব কীভাবে একটি গ্রাফকে উপস্থাপন করতে হয়।

Scanned by CamScanner

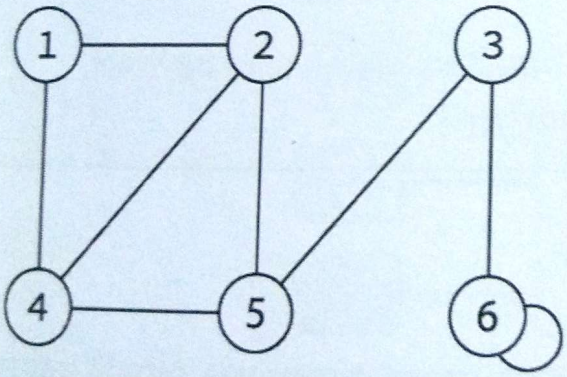
অধ্যায় ২ – গ্রাফ উপস্থাপন

ছবিতে আমরা বৃত্ত আর সরলরেখা দিয়ে সহজেই গ্রাফ দেখিয়ে দিচ্ছি কিন্তু প্রোগ্রামিংয়ের সময় বা অ্যালগরিদম প্রয়োগের সময় আমরা ছবি এঁকে কাজ করতে পারব না, আমাদের কোনো পদ্ধতি জানতে হবে যাতে করে গাণিতিকভাবে একটি গ্রাফকে উপস্থাপন করা যায়, প্রোগ্রামিং করার সময় বিভিন্ন ডেটা স্ট্রাকচার ব্যবহার করে গ্রাফটিকে সংরক্ষণ করা যায়। নিচের দুটি পদ্ধতি খুবই প্রচলিত:

- অ্যাডজেসেন্সি ম্যাট্রিক্স
- অ্যাডজেসেন্সি লিস্ট

২.১ – অ্যাডজেসেন্সি ম্যাট্রিক্স (Adjacency Matrix)

গ্রাফে অ্যাডজেসেন্ট নোড কাকে বলে সেটি আমরা শিখে এসেছি। অ্যাডজেসেন্সি ম্যাট্রিক্সের ধারণাটি খুব সহজ, আমরা একটা 2D অ্যারে বা ম্যাট্রিক্স নিব, এরপর দুটি নোড i, j এর মধ্যে যদি এজ থাকে তাহলে অ্যারের সেই অবস্থানে 1 বসিয়ে দিব। চিত্র ২.১ দেখ।



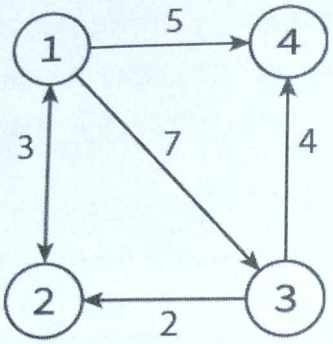
নোড	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

চিত্র ২.১: একটি আনডিরেক্টেড গ্রাফ ও তার অ্যাডজেসেন্সি ম্যাট্রিক্স

এজগুলো ওয়েটেড হতে পারে, যেমন ঢাকা থেকে চট্টগ্রামে একটি এজ দিয়ে বলে দিতে পারে শহর দুটির দূরত্ব 300 কিলোমিটার। তাহলে তোমাকে ম্যাট্রিক্সে ওয়েটও বসাতে হবে।

Scanned by CamScanner

চিত্র ২.১ এর গ্রাফটি বাইডিরেকশনাল বা আনডিরেক্টেড, অর্থাৎ নোড 1 থেকে নোড 2 তে যাওয়া গেলে নোড 2 থেকে নোড 1 এও যাওয়া যাবে। যদি গ্রাফটি ডিরেক্টেড হতো তাহলে এজগুলোর মধ্যে তীরচিহ্ন থাকত। তখনও আমরা আগের মতো করেই ম্যাট্রিক্সে স্টোর করতে পারব। চিত্র ২.২ এ ডিরেক্টেড ওয়েটেড গ্রাফের অ্যাডজেসেন্সি ম্যাট্রিক্সের উদাহরণ দেখানো হয়েছে।



নোড	1	2	3	4
1	inf	3	7	5
2	3	inf	inf	inf
3	inf	2	inf	4
4	inf	inf	inf	inf

চিত্র ২.২: একটি ডিরেক্টেড গ্রাফ ও তার অ্যাডজেসেন্সি ম্যাট্রিক্স

যদি দুটি নোডের মধ্যে কোনো এজ না থাকে তাদেরকে ইনফিনিটি (Infinity) বা অনেক বড় একটি সংখ্যা দিয়ে দেখানো হয়। একটি ব্যাপার লক্ষ কর, গ্রাফ আনডিরেক্টেড হলে ম্যাট্রিক্সটি সিমেট্রিক (Symmetric) হয়ে যায়, অর্থাৎ, $mat[i][j] = mat[j][i]$ হয়ে যায়।

লক্ষ কর

আনডিরেক্টেড গ্রাফের অ্যাডজেসেন্সি ম্যাট্রিক্স সিমেট্রিক (Symmetric) হয়, অর্থাৎ,
$$mat[i][j] = mat[j][i]$$

অ্যাডজেসেন্সি ম্যাট্রিক্স ব্যবহার করার সমস্যা

1. মেমোরি অনেক বেশি নেয়। Nটি নোডের একটা গ্রাফের জন্য $O(N^2)$ মেমোরি দরকার হবে। গ্রাফের এজ সংখ্যা কম হলেও আমরা এক্ষেত্রে কোনো সুবিধা পাবো না।
2. কোনো একটি নোডের সবগুলো অ্যাডজেসেন্ট নোড খুঁজে বের করতে গ্রাফের Nটি নোডই পরীক্ষা করে দেখতে হয়, গ্রাফে নোডের তুলনায় এজের সংখ্যা কম হলে এভাবে পরীক্ষা করতে অযথা বেশি সময় নষ্ট হবে।
3. ওয়েটেড গ্রাফে দুটি নোডের মধ্যে একাধিক এজ থাকলে ম্যাট্রিক্স দিয়ে সেটি উপস্থাপন করা সম্ভব নয়।

Scanned by CamScanner

অ্যাডজেসেন্সি ম্যাট্রিক্স ব্যবহার করার সুবিধা

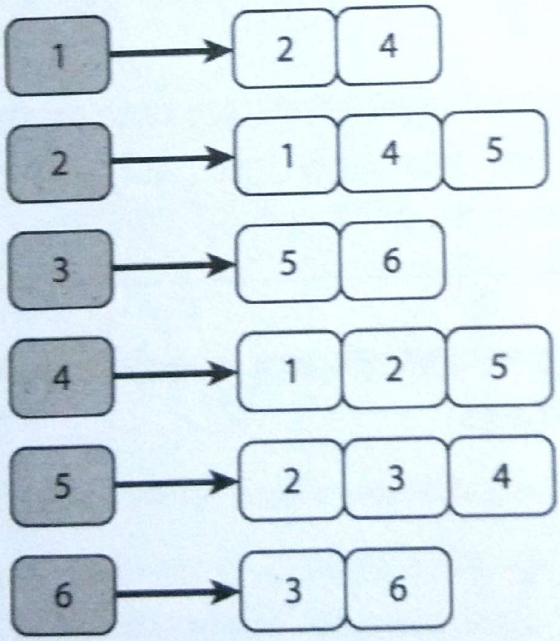
1. দুটি নোড u, v এর মধ্যে কোনো এজ আছে কি না অথবা সেই এজের ওয়েট কত সেটি খুব সহজে $O(1)$ এ পরীক্ষা করা যায়।
2. একটি সাধারণ 2D অ্যারে ব্যবহার করে সহজেই ম্যাট্রিক্স বানানো যায়।

প্রোগ্রামিং টিপস

আনওয়েটেড গ্রাফের ক্ষেত্রে অ্যাডজেসেন্সি ম্যাট্রিক্স হিসাবে ইন্টিজারের বদলে বুলিয়ান 2D অ্যারে ব্যবহার করলে মেমোরি কম খরচ হবে।

২.২ – অ্যাডজেসেন্সি লিস্ট (Adjacency List)

চিত্র ২.১ আরেকবার দেখো। এবার আমরা বাজার করার তালিকার মত একটা তালিকা বানাব।



চিত্র ২.৩: চিত্র ২.১ এর গ্রাফের অ্যাডজেসেন্সি লিস্ট

চিত্র ২.৩ এ যা দেখতে পাচ্ছো সেটিই অ্যাডজেসেন্সি লিস্ট, কোন নোডের সঙ্গে কোন নোড যুক্ত আছে সেটির একটি তালিকা। এটিই হলো অ্যাডজেসেন্সি লিস্ট। প্রতিটি নোডের একটি করে তালিকা থাকবে যেখানে লেখা থাকবে অ্যাডজেসেন্ট নোডগুলোর নাম।

Scanned by CamScanner

অ্যাডজেসেন্সি লিস্ট ব্যবহার করার সুবিধা

1. যতগুলো এজ আছে ঠিক তত মেমোরি ব্যবহার করেই আমরা গ্রাফটি সেভ করে ফেলতে পারব। ম্যাট্রিক্সে গ্রাফ সেভ করার জন্য মেমোরি কমপ্লেক্সিটি ছিল $O(N^2)$, এখানে কমপ্লেক্সিটি $O(E)$। তবে একটি গ্রাফে এজ সংখ্যা E কিন্তু N^2টি হতে পারে, সেক্ষেত্রে মেমোরি একই লাগবে। এজ যত কম হবে, লিস্টে মেমোরি তত কম লাগবে।
2. একটি নোডের সঙ্গে অ্যাডজেসেন্ট নোড কারা কারা সেটি দেখার জন্য গ্রাফের প্রতিটি নোড পরীক্ষা করার দরকার নেই, তালিকা দেখেই বলা যাবে।

অ্যাডজেসেন্সি লিস্ট ব্যবহার করার অসুবিধা

1. দুটো নোড u, v এর মধ্যে এজ আছে কি না যাচাই করতে লিস্টের ওপর লুপ চালাতে হবে। ম্যাট্রিক্সের মতো $O(1)$ এ বের করা সম্ভব নয়।

প্রোগ্রামিং টিপস

সি++ এ ভেক্টর বা জাভাতে অ্যারেলিস্ট ব্যবহার করে অ্যাডজেসেন্সি লিস্ট তৈরি করা যেতে পারে। গ্রাফটি যদি ওয়েটেড হয় তাহলে একটি পেয়ার (Pair) এর লিস্ট তৈরি করা যেতে পারে যেখানে নোডের নামের সঙ্গে ওয়েটও রাখা থাকবে।

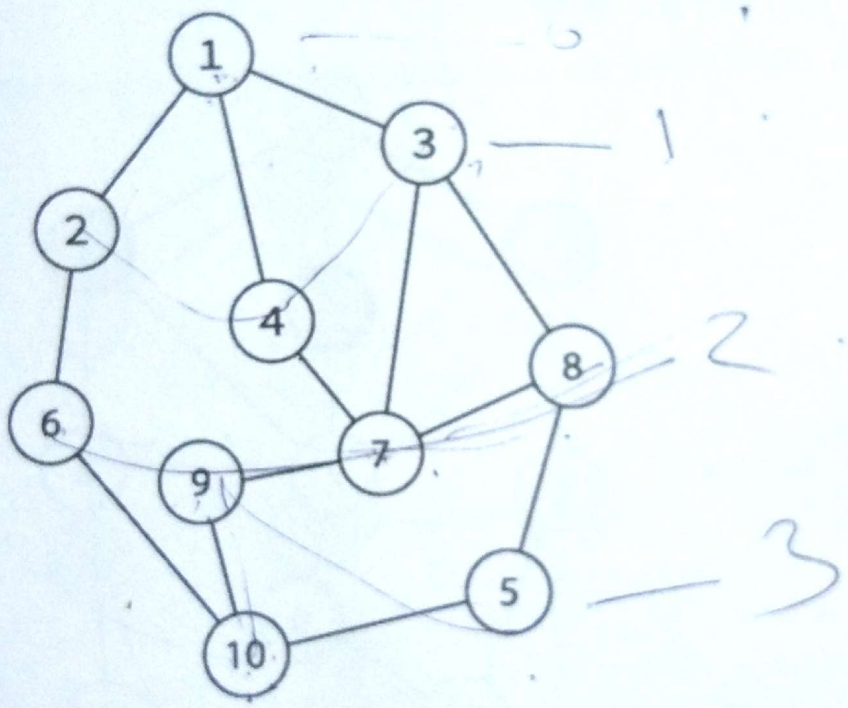
গ্রাফ উপস্থাপন করা শেখার পর এখন তুমি প্রথম অ্যালগরিদম ব্রেডথ ফার্স্ট সার্চ (Breadth First Search) শেখার জন্য প্রস্তুত।

Scanned by CamScanner

অধ্যায় ৩ – ব্রেডথ ফার্স্ট সার্চ (Breadth First Search)

৩.১ – অ্যালগরিদমের বিবরণ

গ্রাফের মধ্যে কোনো কিছু খুঁজে বের করার জন্য অনেকগুলো অ্যালগরিদমের একটি হলো ব্রেডথ ফার্স্ট সার্চ (Breadth First Search) বা বিএফএস (BFS)। এই অ্যালগরিদমটির সব থেকে প্রচলিত ব্যবহার হলো কোনো একটি নোড থেকে অন্য যেকোনো নোডে যাওয়ার সংক্ষিপ্ততম পথ খুঁজে বের করা। তবে এটি শুধুমাত্র আনওয়েটেড গ্রাফে কাজ করে, তারমানে প্রতিটি এজের কস্ট (Cost) হতে হবে 1। তাহলে যে পথে এক নোড থেকে আরেক নোডে সব থেকে কম সংখ্যক এজ লাগবে সেটিই আনওয়েটেড গ্রাফের সংক্ষিপ্ততম পাথ বা শর্টেস্ট পাথ (Shortest Path)।



চিত্র ৩.১: একটি আনওয়েটেড আনডিরেক্টেড গ্রাফ

চিত্র ৩.১ এর গ্রাফে নোড 1 থেকে নোড 8 এ যাওয়ার শর্টেস্ট পাথ হলো [1→3→8]। দুটি নোডের মধ্যে একাধিক শর্টেস্ট পাথ থাকতে পারে, তবে ব্রেডথ ফার্স্ট সার্চ দিয়ে আমরা একটিমাত্র পাথ পাব। ব্রেডথ ফার্স্ট সার্চে একটি সোর্স নোড বলে দিতে হয়, অ্যালগরিদমের শেষে সেই নোড থেকে অন্য সব নোডের শর্টেস্ট পাথ পাওয়া যায়।

প্রতিটি নোডের জন্য আমরা একটি "লেভেল" সংজ্ঞায়িত করব।

- সোর্স নোডের লেভেল 0।

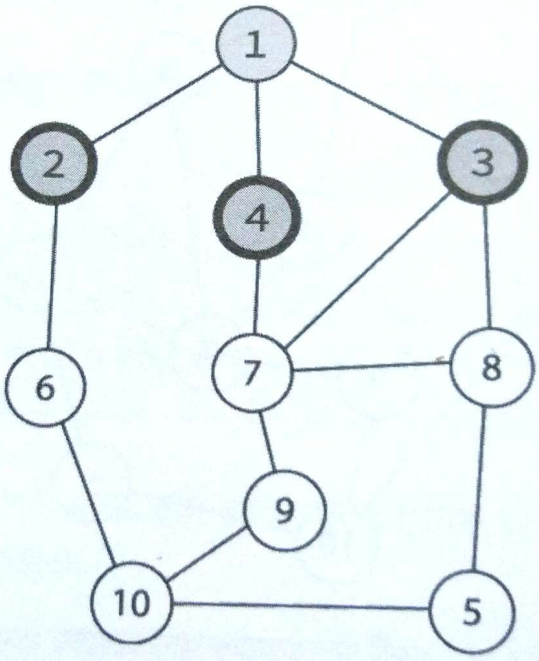
Scanned by CamScanner

- লেভেল 0 নোড থেকে যেসব নতুন নোডে একটি এজ ব্যবহার করে সরাসরি যাওয়া যায় তারা লেভেল 1 নোড।
- লেভেল 1 নোড থেকে যেসব নতুন নোডে একটি এজ ব্যবহার করে সরাসরি যাওয়া যায় তারা লেভেল 2 নোড।

একটি নোডে কখনওই একাধিকবার আমরা ফেরত যাব না, তাই বলা হয়েছে "নতুন নোড"।

আমরা অ্যালগরিদমটি শিখব একটা উদাহরণ দিয়ে। ধর আমরা নোড 1 থেকে নোড 10 এ যেতে চাই। প্রথমে আমরা সোর্স নোড ধরলাম নোড 1 কে। 1 তাহলে একটি "লেভেল 0" নোড। 1 কে ভিজিটেড (Visited) চিহ্নিত করি।

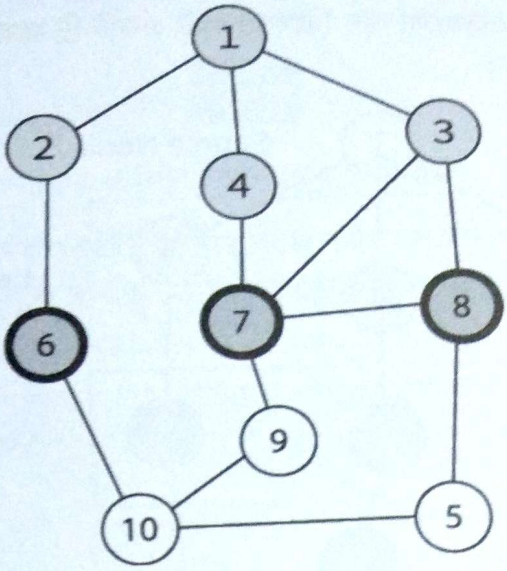
নোড 1 থেকে সরাসরি যাওয়া যায় নোড 2, নোড 3 ও নোড 4 এ। তাহলে নোড 2, নোড 3 ও নোড 4 হলো "লেভেল 1" নোড। এবার সেগুলোকে আমরা ভিজিটেড চিহ্নিত করি এবং সেগুলো নিয়ে কাজ করি। চিত্র ৩.২ দেখো।



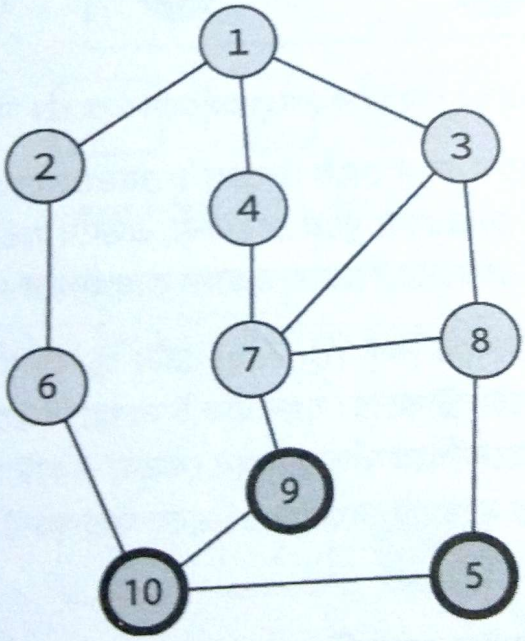
চিত্র ৩.২: নোড 1 থেকে সরাসরি যাওয়া যায় নোড 2, নোড 3 ও নোড 4 এ

গাঢ় ধূসর রঙের নোডগুলো নিয়ে আমরা এখন কাজ করব। একই নোডে একাধিকবার কখনও যাবোনা। নোড 2, নোড 3 ও নোড 4 থেকে সরাসরি যাওয়া যায় নোড 6, নোড 7 ও নোড 8 এ (চিত্র ৩.৩)। লক্ষ কর, যে নোডকে যত নম্বর লেভেলে পাচ্ছি, সোর্স থেকে তার শর্টেস্ট পথের দৈর্ঘ্য ঠিক তত। যেমন লেভেল 3 তে নোড 8 কে পেয়েছি তাই সোর্স থেকে 8 এর দূরত্ব 2। গাঢ় ধূসর রং দিয়ে বোঝানো হয়েছে আমরা এখন ওগুলো নিয়ে কাজ করছি। আমরা এখনো নোড 10 এ পৌঁছাইনি তাই পরের নোডগুলো ভিজিট করে ফেলি।

Scanned by CamScanner



চিত্র ৩.৩: নোড 2, নোড 3 ও নোড 4 থেকে সরাসরি যাওয়া যায় নোড 6, নোড 7 ও নোড 8 এ

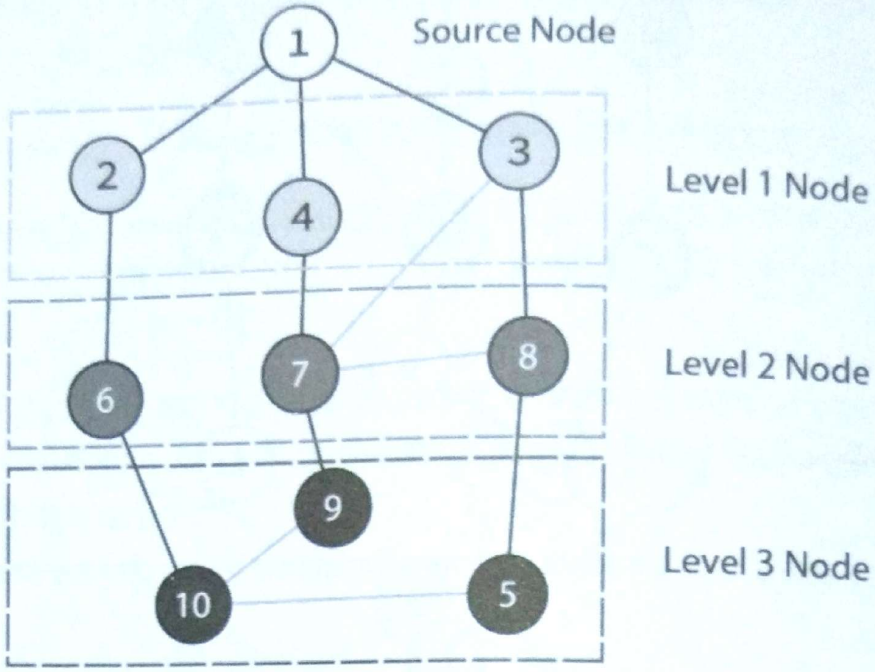


চিত্র ৩.৪: নোড 6, নোড 7 ও নোড 8 থেকে সরাসরি যাওয়া যায় নোড 5, নোড 9 ও নোড 10 এ

আমরা দেখতে পাচ্ছি 2টি লেভেল পার হয়ে 3 নম্বর লেভেলে আমরা নোড 10 কে পাচ্ছি। তাহলে নোড 1 থেকে নোড 10 এর শর্টেস্ট পথ 3। লেভেল-বাই-লেভেল গ্রাফটিকে সার্চ করে আমরা শর্টেস্ট পথ বের করলাম। যেসব এজগুলো আমরা ব্যবহার করিনি সেগুলোকে বাদ দিয়ে গ্রাফটিকে চিত্র ৩.৫ এর মতো করে আঁকতে পারি। যেসব এজ ব্যবহার করিনি ছবিতে সেগুলো

Scanned by CamScanner

হালকা করে দিয়েছি, এই এজগুলো বাদ দিলে গ্রাফটি একটি ট্রি হয়ে যায়। এই ট্রিটিকে বলা হয় বিএফএস ট্রি (BFS Tree)।



চিত্র ৩.৫: চিত্র ৩.১ এর গ্রাফ থেকে নোড 1 কে সোর্স নোড ধরে আঁকা বিএফএস ট্রি

তার মানে আমাদের কাজ হলো সোর্স থেকে লেভেল 1 নোডগুলোতে যাওয়া, তারপর লেভেল নোডগুলো থেকে লেভেল 2 নোডগুলো খুঁজে বের করা, এভাবে যতক্ষণ না গন্তব্যে পৌঁছে যাচ্ছি অথবা সব নোড ভিজিট করা শেষ হয়ে গিয়েছে ততক্ষণ কাজ চলতে থাকবে।

আমরা বিএফএসের কোড লিখতে কিউ (Queue) ডেটা স্ট্রাকচার ব্যবহার করব। কিউ হলো ঠিক বাসের লাইনের মতো ডেটা স্ট্রাকচার। যখন একটি সংখ্যা কিউতে যোগ করা হয় তখন সেটি আগের সবগুলো সংখ্যার পেছনে গিয়ে দাঁড়ায়, যখন কোনো একটি সংখ্যা কিউ থেকে বের করে ফেলা হয় তখন সবার প্রথমের সংখ্যাটি নেওয়া হয়। একে বলা ফার্স্ট ইন ফার্স্ট আউট (First In First Out)।

বিএফএসে লেভেল 1 থেকে যখন কয়েকটি নতুন লেভেল 2 নোড পাব সেগুলোকে কিউতে লাইনে অপেক্ষা করিয়ে রাখব আর সবসময় কিউয়ের প্রথম নোডটি নিয়ে কাজ করব। তাহলে কম লেভেলের নোডগুলো সবসময় পেছনের দিকে থাকবে, আমরা ছোট লেভেলের নোডগুলো নিয়ে কাজ করতে করতে এগিয়ে যাবো। উপরের গ্রাফের জন্য এটা আমরা সিমুলেশন (Simulation) করে দেখি।

প্রথমে কিউতে সোর্স নোড পুশ করব। নোড 1 এর লেভেল হবে 0 বা `লেভেল[1] = 0`। এবার বিএফএস শুরু করব।

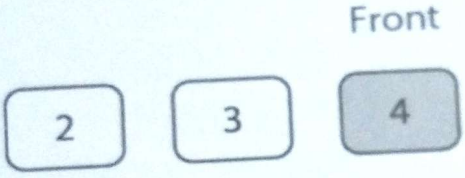
Scanned by CamScanner

চিত্র ৩.৬: কিউতে সোর্স নোড পুশ করা হল

প্রথমে কিউয়ের সবার সামনের নোডটিকে নিয়ে কাজ করব। সবার সামনে আছে নোড 1, সেখান থেকে যাওয়া যায় নোড 4, নোড 3 ও নোড 2 তে। নোড 4, নোড 3 ও নোড 2 তে এসেছি নোড 1 থেকে, তাহলে

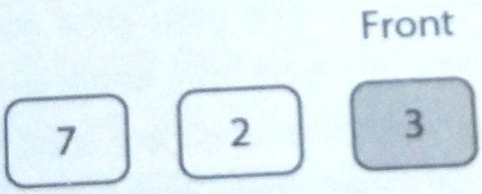
```
লেভেল[4] = লেভেল[1] + 1 = 1
লেভেল[3] = লেভেল[1] + 1 = 1
লেভেল[2] = লেভেল[1] + 1 = 1
```

নোড 1 কে কিউ থেকে ফেলে দিয়ে নোড 4, নোড 3 ও নোড 2 কে কিউতে পুশ করে রাখি।



চিত্র ৩.৭: নোড 1 কে কিউ থেকে ফেলে দিয়ে নোড 4, নোড 3 ও নোড 2 কে কিউতে পুশ করা হল

এবার নোড 4 নিয়ে কাজ করি। নোড 4 থেকে যাওয়া যায় নোড 7 এ। তাহলে আমরা বলতে পারি `লেভেল[7] = লেভেল[4] + 1 = 2`। নোড 4 কে কিউ থেকে ফেলে দিয়ে নোড 7 কে কিউতে পুশ করি।

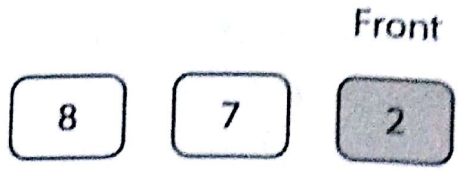


চিত্র ৩.৮: নোড 4 কে কিউ থেকে ফেলে দিয়ে নোড 7 কে কিউতে পুশ করা হল

এবার নোড 3 নিয়ে কাজ করি। নোড 3 থেকে নোড 7 ও নোড 8 এ যাওয়া যায়। নোড 7 কে এরই মধ্যে কিউতে নিয়েছি (অর্থাৎ, ভিজিটেড)। শুধু নোড 8 পুশ করতে হবে। `লেভেল[8] = লেভেল[3] + 1 = 2`।

এভাবে যতক্ষণ না কিউ খালি হচ্ছে ততক্ষণ কাজ চলতে থাকবে। `লেভেল[ ]` অ্যারের মধ্যে আমরা পেয়ে যাবো সোর্স নোড থেকে সবগুলো নোডের দূরত্ব!

Scanned by CamScanner



চিত্র ৩.৯: নোড 4 কে কিউ থেকে ফেলে দিয়ে নোড 7 কে কিউতে পুশ করা হল

```
procedure BFS(G, source):
   Q ← queue(), level[] ← infinity
   Q.enqueue(source)
   level[source] ← 0

   while Q is not empty
     u ← Q.pop()

     for all edges from u to v in G.adjacentEdges(u) do
       if level[v] = infinity:
         level[u] ← level[v] + 1;
         Q.enqueue(v)
       end if
     end for

   end while

   return distance
```

সুডোকোড ৩.১

অনেক সময় পাথের দৈর্ঘ্যের পাশাপাশি পাথে কোন কোন নোড আছে সেটিও দরকার হতে পারে। সেক্ষেত্রে কিউ তে নতুন নোড পুশ করার সময় আমরা $previous[u] = v$ লিখতে পারি যার মানে হলো নোড u তে এসেছি নোড v থেকে। এখন আমরা যেকোনো নোড থেকে $previous$ অ্যারে ব্যবহার করে সোর্স নোডে ফিরে গেল পথটি পেয়ে যাব।

কমপ্লেক্সিটি

প্রতিটি নোডে একবার করে গিয়েছি, প্রতিটি এজে একবার গিয়েছি। তাহলে কমপ্লেক্সিটি হবে $O(V + E)$ যেখানে V হলো নোড সংখ্যা এবং E হলো এজ সংখ্যা।

Scanned by CamScanner

2D গ্রিডে বিএফএস

কখনো কখনো 2D গ্রিডে বিএফএস অ্যালগরিদম প্রয়োগ করা দরকার হতে পারে। যেমন, একটি দাবার বোর্ডে একটি ঘোড়া আর একটি রাজা আছে। ঘোড়াটি সর্বনিম্ন কয়টি মুভ (Move) বা চাল দিয়ে রাজার ঘরে পৌঁছাতে পারবে? আগে নোডকে আমরা প্রকাশ করছিলাম একটিমাত্র সংখ্যা দিয়ে। এখন নোডকে প্রকাশ করতে হবে দুটি সংখ্যা দিয়ে, রো (Row) নম্বর এবং কলাম (Column) নম্বর। প্রতিটি নোডের অ্যাডজেসেন্ট নোড হবে ঘোড়াটি একটি চাল দিয়ে যেসব ঘরে যেতে পারে সেসব ঘর। বাকি সব কিছু একইরকম থাকবে।

প্রোগ্রামিং টিপস

2D গ্রিডে কাজ করার জন্য তুমি একটি ক্লাস বানিয়ে ফেলতে পারো যেখানে দুটি ইন্টিজার থাকবে যা রো এবং কলাম নির্দেশ করবে। সি++ এ কাজটা করা যায় এভাবে: "`struct node{int r,c;};`"।

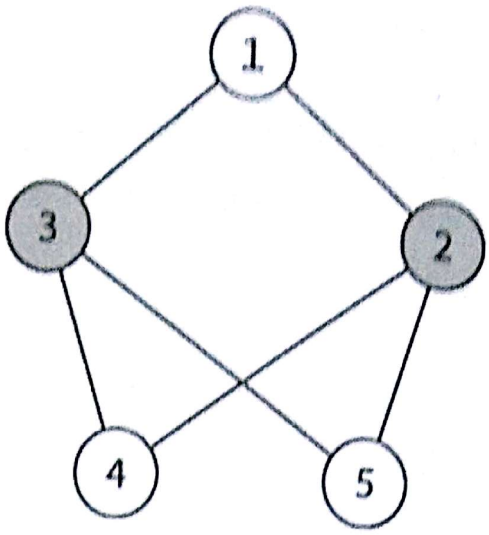
৩.২ – বাইকালারিং (Bicoloring)

একটা গ্রাফের সবগুলো নোডকে শুধুমাত্র দুটি রঙ দিয়ে রঙ করা গেলে সেটিকে বাইকালারেবল (Bicolorable) গ্রাফ বলে, শর্ত হলো দুটি অ্যাডজেসেন্ট নোডের রঙ একই হতে পারবে না।

একটি গ্রাফ বাই-কালারেবল নাকি সেটি বিএফএস অ্যালগরিদম বা ডিএফএস অ্যালগরিদম দিয়ে খুব সহজেই পরীক্ষা করা যায়। এক নোড থেকে অন্য নোডে যাওয়ার সময় আগে দেখতে হবে নোডটির রঙ কী। যদি এখনও রঙ করা না হয়ে থাকে তাহলে বর্তমান নোডে যে রঙ আছে তার বিপরীত রঙ ব্যবহার করতে হবে নতুন নোডে। আর যদি দেখ যে আগেই রঙ করা আছে তাহলে দেখবে বর্তমান নোডের সঙ্গে রঙটি মিলে গিয়েছে কি না, মিলে গেলে গ্রাফটা বাই-কালারেবল নয়।

একটি গ্রাফকে সর্বনিম্ন কয়টি রঙ ব্যবহার করে রঙ করা সম্ভব যাতে পাশাপাশি দুটি নোডের রঙ একই না হয়? এটি একটি বিখ্যাত সমস্যা যার পলিনোমিয়াল কোনো সমাধান খুঁজে পাওয়া যায় নি।

Scanned by CamScanner



চিত্র ৩.৬: বাইকালারেবল গ্রাফ

লক্ষ কর

একটি গ্রাফে যদি বিজোড় দৈর্ঘ্যের কোনো সাইকেল থাকে তাহলে সেই গ্রাফটি বাইকালার করা সম্ভব নয়!

ব্রেডথ ফার্স্ট সার্চ অ্যালগরিদম ব্যবহার করে বাইকালারিংয়ের সুডোকোড নিচে দেওয়া হলো।

```
procedure Bicoloring(G, source):
  Q ← queue()
  Q.enqueue(source)
  color[] ← white
  color[source] ← red

  while Q is not empty
    u ← Q.pop()

    for all edges from u to v in G.adjacentEdges(u) do
      if color[v] = white

        if color[u] = red
          color[v] ← blue
        else
          color[v] ← red
        end if
```

Scanned by CamScanner

```
        Q.enqueue(v)

      end if

      if color[u] = color[v]
        return false
      end if
    end for

  end while

  return true
```

সুডোকোড ৩.২

Scanned by CamScanner

অধ্যায় ৪ – ডায়াক্সট্রা অ্যালগরিদম (Dijkstra Algorithm)

বিএফএস অ্যালগরিদমের সমস্যা হলো গ্রাফে ওয়েট (weight) থাকলে এই অ্যালগরিদম কাজ করে না। ওয়েটেড গ্রাফের জন্য অনেকগুলো অ্যালগরিদম আছে, আমরা প্রথমেই শিখব ডায়াক্সট্রা অ্যালগরিদম।

৪.১ – পাথ রিলাক্সেশন

ডায়াক্সট্রা অ্যালগরিদম শুরু করার আগে আমরা পাথ রিল্যাক্সেশন (Path Relaxation) নামের একটা ছোট্ট জিনিসের সঙ্গে পরিচিত হই। ধরো সোর্স নোড থেকে প্রতিটি নোডের ডিসটেন্স রাখা হয়েছে $distance[]$ অ্যারেতে। যেমন $distance[3]$ মানে হলো সোর্স থেকে বিভিন্ন এজ পার হয়ে নোড 3 তে আসতে মোট $distance[3]$ ডিসটেন্স লেগেছে। যদি ডিসটেন্স জানা না থাকে তাহলে ইনফিনিটি অর্থাৎ অনেক বড় একটা মান রেখে দিব। আর $cost[u][v]$ তে রাখা আছে $[u - v]$ এজের ওয়েট।

ধরো তুমি বিভিন্ন জায়গা ঘুরে ফার্মগেট থেকে টিএসসিতে গেলে 10 মিনিটে, আবার ফার্মগেট থেকে কার্জন হলে গেলে 25 মিনিটে। এক্ষেত্রে ফার্মগেট হলো সোর্স। তাহলে তুমি নিচের তথ্যটি জানো,

$$distance[\text{টিএসসি}] = 10,$$

$$distance[\text{কার্জন হল}] = 25$$

এখন তুমি দেখলে টিএসসি থেকে 7 মিনিটে কার্জনে চলে যাওয়া যায়,

$$cost[\text{টিএসসি}][\text{কার্জন হল}] = 7$$

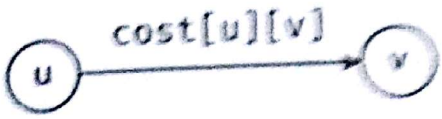
তাহলে তুমি 25 মিনিটের জায়গায় মাত্র $10 + 7 = 17$ মিনিটে কার্জন হলে যেতে পারবে। যেহেতু তুমি দেখেছ,

$$distance[\text{টিএসসি}] + cost[\text{টিএসসি}][\text{কার্জন হল}] < distance[\text{কার্জন হল}]$$

তাই তুমি এখন এই নতুন রাস্তা দিয়ে কার্জন হলে গিয়ে, $distance[\text{কার্জন হল}] = distance[\text{টিএসসি}] + cost[\text{টিএসসি}][\text{কার্জন হল}]$ বানিয়ে দিতেই পারো!

চিত্র ৪.১ সেটিই বলছে।

Scanned by CamScanner




```
if d[u] + cost[u][v] < d[v] :
    d[v] = d[u] + cost[u][v]
```

চিত্র ৪.১: ক্ষুদ্রতর মান দিয়ে *distance* এর মান আপডেট করা

আমরা নোড u থেকে নোড v তে যাবো যদি $distance[u] + cost[u][v] < distance[v]$ হয়। আর $distance[v]$ কে আপডেট করে $distance[v] = distance[u] + cost[u][v]$ বানিয়ে দিব। ভবিষ্যতে যদি কার্ডিন হয় অন্য রাস্তা দিয়ে আরও কম সময়ে যেতে পারি তখন সেই রাস্তা এভাবে তুলনা করে আপডেট করে দিব। ব্যাপারটা অনেকটা এরকম,

```
if distance[u] + cost[u][v] < distance[v]
  distance[v] ← distance[u] + cost[u][v]
end if
```

উপরের অংশটি যদি বুঝে থাকো তাহলে ডায়াক্সট্রা অ্যালগরিদম বোঝার কাজ অর্ধেক হয়ে গেছে। না বুঝে থাকলে আবার পড়।

৪.২ – অ্যালগরিদমের বিবরণ

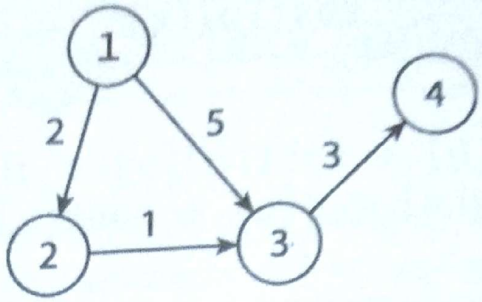
বিএফএস অ্যালগরিদম নিশ্চয়ই তুমি ভালো করে বোঝ। বিএফএস অ্যালগরিদমে আমাদের একটি নোডে কখনও দ্বিতীয়বার যাওয়ার দরকার হয়নি, আমরা প্রতিবার দেখেছি একটি নোড ভিজিটেড কি না, যদি ভিজিটেড না হয় তাহলে সেই নোডটিকে কিউতে পুশ করে দিয়েছি এবং ডিসটেন্স 1 বাড়িয়ে দিয়েছি। ডায়াক্সট্রাতে আমরা একইভাবে কিউতে নোড রাখব তবে ভিজিটেড দিয়ে আপডেট না করে নতুন এজকে "রিল্যাক্সিং (Relaxing)" বা আপডেট করব উপরের পদ্ধতিতে। চিত্র ৪.২ এর গ্রাফটি দেখো।

ধরে নেই সোর্স হলো 1 নম্বর নোড। তাহলে,

$$distance[1] = 0,$$

$distance[2] = distance[3] = distance[4] = infinity$ (অনেক বড় একটি সংখ্যা)

Scanned by CamScanner



চিত্র ৪.২: একটি ওয়েটেড ডিরেক্টেড গ্রাফ

কিন্তু নোড 5 তো নেড 3 এ যাওয়ার শর্টেস্ট ডিসটেন্স নয়! আমরা বিএফএস অ্যালগরিদমে দেখেছি একটি নোড একবারের বেশি আপডেট হয়না, সেই বৈশিষ্ট্য বা প্রপার্টি এখানে কাজ করছে না। নোড 2 থেকে $[2-3]$ এজ ধরে এগিয়ে আবার আপডেট করলে তখন $distance[3]$ তে $distance[2]+1=3$ পাব। তাহলে আমরা দেখলাম এক্ষেত্রে একটি নোড একাধিকবার আপডেট হতে পারে।

আমরা তাহলে আগের বিএফএসের কোডের আপডেট অংশ একটু পরিবর্তন করি যাতে একটি নোড বার বার আপডেট হতে পারে,

```
procedure BFSmodified(G,source):
  Q ← queue()
  distance[] ← infinity
  Q.enqueue(source)
  distance[source] ← 0

  while Q is not empty
    u ← Q.pop()

    for all edges from u to v in G.adjacentEdges(v) do
      if distance[u] + cost[u][v] < distance[v]
        distance[v] ← distance[u] + cost[u][v]
        Q.enqueue(v)
      end if
    end for
  end while

return distance
```

সুডোকোড ৪.১

Scanned by CamScanner

আমরা ঠিক আগের বিএফএস অ্যালগরিদমে এর কোডেই শুধুমাত্র ওয়েট (বা কস্ট) বসিয়ে বার বার আপডেট করছি! এই কোড তোমাকে সোর্স থেকে প্রতিটি নোডের শর্টেস্ট পাথ বের করে দিবে কিন্তু একটু বুদ্ধি খাটিয়ে আরও কম কমপ্লেক্সিটিতে আমরা কাজটি করতে পারি।

বিএফএস অ্যালগরিদমে আমরা যখন একটি নোড থেকে অনেকগুলো নোডে যাচ্ছি তখন সেই নোডগুলো থেকে আবার নতুন করে কাজ করার সময় "আগে আসলে আগে পাবে" (First in first out queue) ভিত্তিতে কাজ করছি। যেমন উপরের গ্রাফে নোড 1 থেকে আগে নোড 3 এ এবং তারপর নোড 2 তে গেলে আগে 3 নিয়ে কাজ করছি, এরপর 2 নিয়ে কাজ করছি।

ভালো করে দেখো এখানে কী সমস্যাটি হচ্ছে। নোড 3 নিয়ে আগে কাজ করলে আমরা নোড 4 এর ডিসটেন্সকে আপডেট করে দিচ্ছি 8 $(= 5 + 3)$ হিসাবে পরবর্তীতে যখন নোড 2 দিয়ে নোড 3 কে আবার আপডেট করা হচ্ছে তখন নোড 3 এর ডিসটেন্স হয়ে গিয়েছে 3, এবার নোড 4 এর ডিসটেন্সকে আবার আপডেট করছি $(3 + 3 =)$ 6 হিসাবে। নোড 4 কে মোট দুইবার আপডেট করা লাগলো।

বিজ্ঞানী ডায়াক্সট্রা (Edsger W. Dijkstra) চিন্তা করলেন যদি এই "আগে আসলে আগে পাবেন" ভিত্তিতে কাজ না করে সবচেয়ে কাছের নোডগুলোকে আগে প্রসেসিং করি তাহলে অনেক কম সংখ্যকবার আপডেট করা লাগে। আমরা যদি নোড 2 কে নিয়ে আগে কাজ করতাম তাহলে নোড 3 আগেই আপডেট হয়ে যেত এবং নোড 4 কে একবার আপডেট করেই শর্টেস্ট ডিসটেন্স পেয়ে যেতাম! এটা তুমি হাতে কলমে সিমুলেশন করে দেখতে পারো।

> যদি "আগে আসলে আগে পাবেন" ভিত্তিতে কাজ না করে সবচেয়ে কাছের নোডগুলোকে আগে প্রসেসিং করি তাহলে অনেক কম সংখ্যকবার আপডেট করা লাগে।

আইডিয়াটি হলো যেকোনো সময় কিউতে যতগুলো নোড থাকবে তাদের মধ্যে যেটি সোর্স থেকে সবথেকে কাছে সেটি নিয়ে আগে কাজ করব। এজন্যই আমরা কিউয়ের জায়গায় বসিয়ে দিব একটি প্রায়োরিটি কিউ যে কিউতে নোড পুশ করার সঙ্গে সঙ্গে কাছের নোডটিকে সামনে এনে দিবে। পার্থক্য হলো আগে খালি নোড নম্বর পুশ করেছি, এখন বর্তমান ডিসটেন্স অর্থাৎ $distance[u]$ এর মানটিও পুশ করতে হবে।

```
1 procedure dijkstra(G, source):
2   Q ← priority_queue()
3   distance[] ← infinity
4   Q.enqueue(source)
```


Scanned by CamScanner

```
distance[source] ← 0

while Q is not empty
  u ← nodes in Q with minimum distance[]
  remove u from the Q

  for all edges from u to v in G.adjacentEdges(v) do
    if distance[u] + cost[u][v] < distance[v]
      distance[v] ← distance[u] + cost[u][v]
      Q.enqueue(v)
    end if
  end for

end while

return distance
```

সুডোকোড ৪.২

প্রোগ্রামিং টিপস

উপরের সুডোকোডে সোর্স থেকে বাকি সব নোডের দূরত্ব বের করা হয়েছে। তুমি যদি শুধু একটি নোডের দুরত্ব বের করতে চাও তাহলে সেটি যখন কিউ থেকে পপ হবে তখনই রিটার্ন করে দিতে পার।

এখানে আগের সুডোকোডের থেকে কয়েক জায়গায় একটু ভিন্নতা আছে। এখানে সাধারণ কিউয়ের জায়গায় প্রায়োরিটি কিউ ব্যবহার করা হয়েছে। কিউ থেকে পপ হওয়ার সময় তাই সোর্স থেকে এখন পর্যন্ত পাওয়া সবচেয়ে কাছের নোডটি পপ হচ্ছে এবং সেটি নিয়ে আগে কাজ করছি।

নেগেটিভ সাইকেল?

একটি কমন ইন্টারভিউ প্রশ্ন হলো গ্রাফে নেগেটিভ সাইকেল থাকলে ডায়াক্সট্রা কেন কাজ করবে না? গ্রাফে নেগেটিভ সাইকেল থাকার অর্থ হল কোনো একটি নোড থেকে শুরু করে তুমি এক বা একাধিক নোড ঘুরে আবার প্রথম নোডে ফিরে আসার মোট কস্ট ঋণাত্মক বা নেগেটিভ। সোর্স

Scanned by CamScanner

থেকে যদি কোনো নেগেটিভ সাইকেলে পৌঁছে যাওয়া যায় সেক্ষেত্রে ডায়াক্সট্রা অসীম সময় ধরে চলতে থাকবে কারণ শর্টেস্ট পাথ শব্দটার তখন কোনো অর্থ নেই, তুমি যত খুশি কস্ট কমাতে পারবে নেগেটিভ সাইকেলে বার বার ঘুরে।

কমপ্লেক্সিটি

বিএফএস অ্যালগরিদমে টাইম কমপ্লেক্সিটি ছিল $O(V + E)$ যেখানে V হলো নোড সংখ্যা আর E হলো এজ সংখ্যা। এখানেও আগের মতোই কাজ হবে তবে প্রায়োরিটি কিউ তে প্রতিবার সর্ট করতে $\log V$ কমপ্লেক্সিটি লাগবে। তাই মোট কমপ্লেক্সিটি $O(V \log V + E)$।

২য় সেরা সংক্ষিপ্ততম পথ

অনেক সময় আমাদের ২য় বা ৩য় শর্টেস্ট পাথ খুঁজে বের করা দরকার হতে পারে। বাস্তবে অনেক সময় দেখা যায় সংক্ষিপ্ত দূরত্বের পথেই সবচেয়ে বেশি ট্রাফিক জ্যাম, তখন ২য় শর্টেস্ট পথে গেলেই হয়তো কম সময় লাগে। এটিও তুমি ডায়াক্সট্রা ব্যবহার করে বের করতে পারো।

হিন্টস

যখন *distance* অ্যারে নিকটতম দূরত্ব দিয়ে আপডেট হবে তখন সেই অ্যারেতে আগে যে দূরত্বটি ছিল সেটা ২য় সেরা!

Scanned by CamScanner

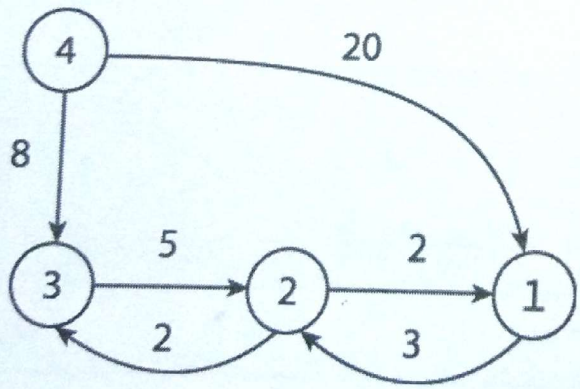
অধ্যায় ৫ – ফ্লয়েড ওয়ার্শল অ্যালগরিদম (Floyd Warshall Algorithm)

আগের অধ্যায়গুলোতে আমরা যে অ্যালগরিদমগুলো শিখেছি সেটি শুধুমাত্র একটি সোর্স থেকে অন্য সব নোডের দূরত্ব বের করে দিতে পারে। এমন হতেই পারে যে আমরা সব নোড থেকে অন্য সব নোডের দূরত্ব জানতে চাই। সেক্ষেত্রে কী করা যেতে পারে?

৫.১ – অ্যালগরিদমের বিবরণ

ফ্লয়েড ওয়ার্শল (Floyd Warshall) অ্যালগরিদম ব্যবহার করে $O(n^3)$ কমপ্লেক্সিটিতে সব নোড থেকে সব নোডের দূরত্ব বের করা যায়। এ ধরনের অ্যালগরিদমকে **"অল পেয়ার শর্টেস্ট পাথ (All pair shortest path) অ্যালগরিদম"** বলা হয়।

আমরা একটি গ্রাফের ওপর কিছু সিমুলেশন করে সহজেই অ্যালগরিদমটি বুঝতে পারি। চিত্র ৫.১ দেখ।



নোড	1	2	3	4
1	0	3	inf	inf
2	2	0	2	inf
3	inf	5	0	inf
4	20	inf	8	0

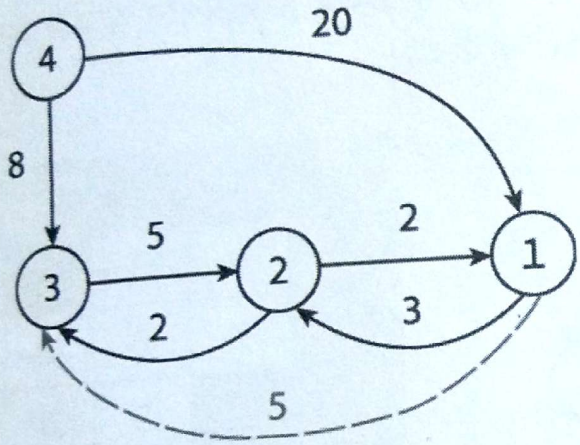
চিত্র ৫.১: একটি ওয়েটেড ডিরেক্টেড গ্রাফ ও তার অ্যাডজেসেন্সি ম্যাট্রিক্স

চিত্র ৫.১ এ চার নোডের একটি ওয়েটেড ডিরেক্টেড গ্রাফ দেখা যাচ্ছে। আর উপরে ডান কোনায় একটি ম্যাট্রিক্স। ম্যাট্রিক্সের u, v তম ঘরে বসানো হয়েছে $[u - v]$ এজের ওয়েট। যাদের মধ্যে সরাসরি এজ নেই, সেসব ঘরে অসীম বা ইনফিনিটি বসিয়ে দেওয়া হয়েছে। আর কোনাকুনি ঘরগুলোতে মান 1 কারণ নিজের বাসা থেকে নিজের বাসাতে যেতে কোনো দূরত্ব অতিক্রম করতে হয় না!

Scanned by CamScanner

এখন মনে করো নোড "2" কে আমরা "মাঝের নোড" হিসেবে ধরলাম। মাঝের নোডকে আমরা বলব k। তার মানে এখন $k = 2$। (আমরা একে একে সব নোডকেই মাঝের নোড হিসাবে ধরবো, এটা যেকোনো অর্ডারে করা যায়)।

এখন যেকোনো এক জোড়া নোড (i, j) নাও। ধরি $i = 1, j = 3$। আমরা নোড k কে মাঝে রেখে নোড i থেকে নোড j তে যেতে চাই। তাহলে আমাদের নোড i থেকে নোড k তে যেতে হবে, তারপর নোড k থেকে থেকে নোড j তে যেতে হবে। কিন্তু লাভ না হলে আমরা এভাবে যাবো কেন? আমরা নোড k কে মাঝে রেখে যাবো কেবল যদি মোট দূরত্ব কমে যায়। নোড 1 থেকে নোড 3 এর বর্তমান দূরত্ব $matrix[1][3] =$ ইনফিনিটি। আর যদি $k = 2$ কে মাঝখানে রাখি তাহলে দূরত্ব দাড়াবে $matrix[1][2] + matrix[2][3] = 3 + 2 = 5$। তার মানে দূরত্ব কমে যাচ্ছে! আমরা গ্রাফটি আপডেট করে দিতে পারি চিত্র ৫.২ মতো করে।



নোড	1	2	3	4
1	0	3	5	inf
2	2	0	2	inf
3	inf	5	0	inf
4	20	inf	8	0

চিত্র ৫.২: নোড 1 থেকে নোড 3 এ যাওয়ার কস্ট আপডেট করা হলো

আমরা নোড 2 কে "মাঝের নোড" হিসেবে ব্যবহার করে নোড 1 থেকে নোড 3 তে গিয়েছি মোট 5 কস্টে। গ্রাফে তাহলে নোড 1 থেকে নোড 3 তে সরাসরি একটি এজ দিয়ে দিতে পারি 5 ওয়েটের। শুরুতে আমাদের নোড 1 থেকে নোড 3 তে কোনো এজ ছিল না। কিন্তু আপডেট করার সময় একটি কাল্পনিক এজ বসিয়ে দিয়েছি।

এখন আমরা কল্পনা করতে পারি নোড 3 থেকে নোড 1 এ 5 কস্টে যাওয়া যায়, যদিও মূল গ্রাফে [1→2→3] এই পথে যেতে হবে।

এখন আবার ধর $i = 2, j = 4$। আর আগের মতোই $k = 2$। এবার $matrix[2][4] =$ ইনফিনিটি। এদিকে $matrix[2][2] + matrix[2][4] = 1 +$ ইনফিনিটি। এবার কিন্তু দূরত্ব

Scanned by CamScanner

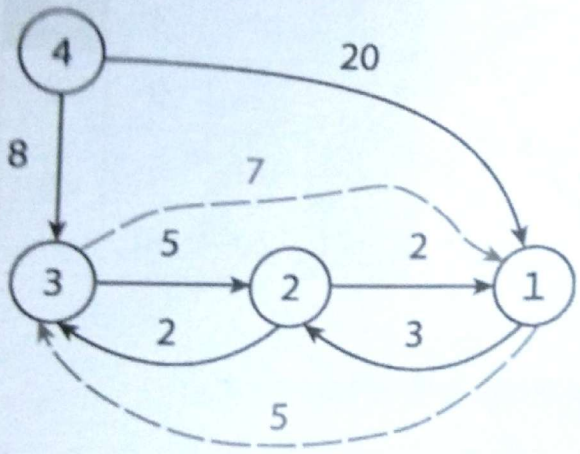
[illegible] আপডেট করার দরকার নেই। নিশ্চয়ই বুঝতে পারছ আপডেটের শর্তটা

```
[illegible] + matrix[k][j] < matrix[i][j]
[illegible] = matrix[i][k] + matrix[k][j]
```

[illegible], আমরা এক লাইনে লিখতে পারি:

```
matrix[i][j] = min(matrix[i][j] , matrix[i][k] + matrix[k][j])
```

এখন $k = 2$ স্থির রেখে আমরা i, j এর সবরকমের কম্বিনেশন (Combination) নিব। তুমি নিজেই চিন্তা করলে দেখবে $k = 2$ এর জন্য $i = 3, j = 1$ এই কম্বিনেশনে আমরা আরেকটি নতুন এজ পাব, বাকি গ্রাফ আগের মতো থাকবে (চিত্র ৫.৩)।



নোড	1	2	3	4
1	0	3	5	inf
2	2	0	2	inf
3	7	5	0	inf
4	20	inf	8	0

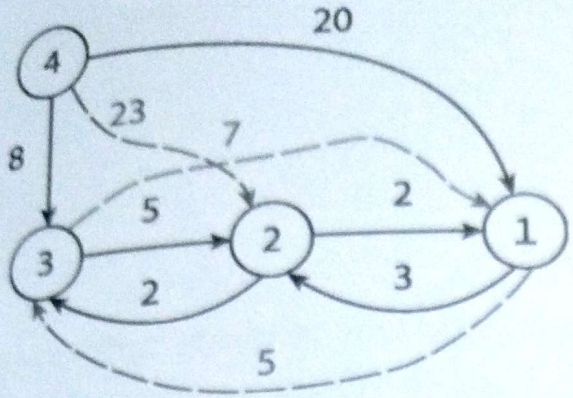
চিত্র ৫.৩: নোড 3 থেকে নোড 1 এ যাওয়ার কস্ট আপডেট করা হলো

এবার আমরা $k = 1$ কে মাঝের নোড হিসাবে চিন্তা করি। মাথা খাটাতে চাইলে নিচে দেখার আগে নিজেই খাতায় এঁকে ফেলতে পারো নতুন এজগুলো। এবার একটিমাত্র এজ যোগ হবে। $i = 4, j = 2$ হলে আমরা নোড 4 থেকে নোড 1 হয়ে 2 নম্বর নোডে যেতে পারি 23 দূরত্ব অতিক্রম করে। তাহলে গ্রাফটি হবে চিত্র ৫.৪ এর মতো।

এবার 3 নম্বর নোডকে মাঝের নোড হিসাবে ধরব। আবারও নিজে চেষ্টা করে তারপর নিচের অংশ দেখ।

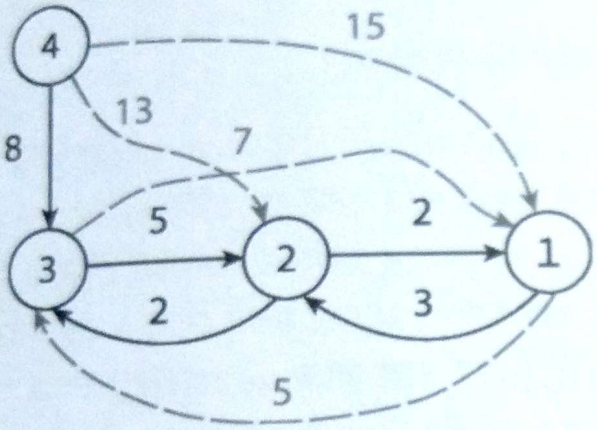
এবার কিছু মজার জিনিস ঘটবে। নোড 4 থেকে নোড 1 এ যেতে আগে লাগছিল 20 দূরত্ব। এখন নোড 3 কে মাঝে রেখে নোড 4 থেকে নোড 1 এ গেলে লাগবে $8 + 7 = 15$ দূরত্ব। একইভাবে নোড 4 থেকে নোড 2 এর 23 দূরত্বের এর পথটা আপডেট হয়ে 13 হয়ে যাবে (চিত্র ৫.৫)।

Scanned by CamScanner



নোড	1	2	3	4
1	0	3	5	inf
2	2	0	2	inf
3	7	5	0	inf
4	20	23	8	0

চিত্র ৫.৪: নোড 4 থেকে নোড 2 এ যাওয়ার কস্ট আপডেট করা হলো



নোড	1	2	3	4
1	0	3	5	inf
2	2	0	2	inf
3	7	5	0	inf
4	15	13	8	0

চিত্র ৫.৫: নোড 4 থেকে নোড 1 এ যাওয়ার কস্ট আপডেট করা হলো

$K = 4$ এর জন্য আর কোন আপডেট হবে না কারণ কোনো নোড থেকে নোড 4 এ যাওয়া যায় না।

এখন আমরা ম্যাট্রিক্স দেখেই বলে দিতে পারছি কোন নোড থেকে কোন নোডে কত দূরত্ব অতিক্রম করে যাওয়া যায়। ইনফিনিটি থাকা মানে নোড i থেকে নোড j তে যাবার কোনো পথ নেই।

ইনপুট থেকে অ্যাডজেসেন্সি ম্যাট্রিক্স বানানোর পর তাহলে অ্যালগরিদমটা হবে এরকম,

```
procedure FloydWarshall(matrix):
  for k from 1 to N do
    for i from 1 to N do
      for j from 1 to N do
        if matrix[i][j] > matrix[i][k] + matrix[k][j]
          matrix[i][j] ← matrix[i][k] + matrix[k][j]
        end if
```


Scanned by CamScanner

```
        end for
      end for
    end for
    return matrix
```

সুডোকোড ৫.১

N হলো গ্রাফে নোড সংখ্যা। শুরুতে দুটি নোডের মধ্যে এজ না থাকলে ম্যাট্রিক্সে সেই স্থানে অবশ্যই ইনফিনিটি থাকতে হবে। কোডে আমরা k এর লুপটি নোড 1 থেকে শুরু করেছি যদিও উদাহরণে আগে নোড 2 নিয়েছি। এটি আসলে যেকোনো ক্রমেই করা যায়, তুমি আগে নোড 1 নিলেও দেখবে অ্যালগরিদম কাজ করবে।

এভাবে তো আমরা শুধু পাথের কস্ট পেলাম, পাথটা কীভাবে পাব?

ধর আমাদের একটি ম্যাট্রিক্স আছে $next[\,][\,]$। এখন $next[i][j]$ দিয়ে আমরা বুঝি নোড i থেকে নোড j তে যেতে হলে পরবর্তী যে নোডে যেতে হবে সেই নোডটিকে। তাহলে একদম শুরুতে সব i, j নোড জোড়ার জন্য $next[i][j] = j$ হবে। কারণ, শুরুতে কোনো "মাঝের নোড" নেই এবং তখনও শর্টেস্ট পাথ বের করা শেষ হয়নি। আমরা যখন নোড k কে মাঝে রেখে নোড i থেকে নোড j তে যাব তখন $matrix[i][j]$ এর সঙ্গে সঙ্গে $next[i][j]$ কেও আপডেট করব।

আমরা যখন নোড k কে মাঝে রেখে নোড i থেকে নোড j তে যাব তখন $matrix[i][j]$ এর সঙ্গে সঙ্গে $next[i][j]$ কেও আপডেট করব।

লক্ষ কর, আমরা কিন্তু মূল গ্রাফে সরাসরি এজ দিয়ে নোড i থেকে নোড k তে নাও যেতে পারি, আমরা শুধু জানি নোড i থেকে নোড j তে যাওয়ার আগে আমাদেরকে নোড i থেকে নোড k তে যেতে হবে। $next[i][k]$ তে রাখা আছে নোড i থেকে নোড k তে যেতে হলে পরবর্তীতে যে নোডে যেতে হবে সেই নোডটি। তাহলে আমরা $next[i][j]$ কে আপডেট করব এভাবে $next[i][j] = next[i][k]$। তার মানে আমরা নোড i থেকে নোড j তে যাওয়ার জন্য পরবর্তীতে কোন নোড এ যেতে হবে সেটি পেয়ে গেলাম।

এখন সুডোকোড দেখি,

```
procedure FloydWarshall(matrix):
  for k from 1 to N do
```


Scanned by CamScanner

```
    for i from 1 to N do
      for j from 1 to N do
        if matrix[i][j] > matrix[i][k] + matrix[k][j]
          matrix[i][j] ← matrix[i][k] + matrix[k][j]
          next[i][j] ← next[i][k]
        end if
      end for
    end for
  end for
  return matrix

procedure FindPath(i, j):
  path ← [i]
  while i ≠ j
    i ← next[i][j]
    path.append(i)
  end while
  return path
```

সুডোকোড ৫.২

findpath ফাংশনে আমরা j কে ফিক্সড রেখে $next$ অ্যারে ধরে আগাচ্ছি যতক্ষণ না j তে পৌঁছাচ্ছি। তাহলে আমরা পেয়ে গেলাম পাথ!

ট্রান্সিটিভ ক্লোজার (Transitive Closure):

ধর আমাদের অ্যাডজেন্সি ম্যাট্রিক্সটা এরকম,

```
matrix[i][j] = 1 যদি নোড i থেকে নোড j তে সরাসরি এজ থাকে
matrix[i][j] = 0 যদি এজ না থাকে
```

এখন আমরা এমন একটি ম্যাট্রিক্স তৈরি করতে চাই যেটি দেখে বলে দেওয়া যাবে নোড i থেকে নোড j তে এক বা একাধিক এজ ব্যবহার করে যাওয়া যায় কি না। আমরা চাইলে উপরের মতো করে 0 এর জায়গায় ইনফিনিটি দিয়ে শর্টেস্ট পাথ বের করে কাজটি করতে পারতাম। কিন্তু এক্ষেত্রে "OR" আর "AND" অপারেশন ব্যবহার আরও দ্রুত কাজটি করা যায়। এখন আপডেটের শর্তটি হয়ে যাবে এরকম,

Scanned by CamScanner

```
matrix[i][j] ← matrix[i][j] || (matrix[i][k] && matrix[k][j])
```

এটির মানে $matrix[i][j]$ তে তখনই 1 বসবে যখন হয় "$matrix[i][j]$ তে 1 আছে" অথবা "$matrix[i][k]$ এবং $matrix[k][j]$" দুটিতেই 1 আছে। তারমানে হয় সরাসরি যেতে হবে অথবা মাঝে একটি নোড k ব্যবহার করে যেতে হবে।

কমপ্লেক্সিটি

3টি নেস্টেড লুপ চলছে নোড সংখ্যার উপর, টাইম কমপ্লেক্সিটি $O(n^3)$। 2D ম্যাট্রিক্স ব্যবহার করায় স্পেস কমপ্লেক্সিটি $O(n^2)$।

প্রশ্ন থাকলো, ফ্লয়েড ওয়ার্শলে k এর লুপটি i, j এর পরে দিলে অ্যালগরিদম ঠিকমত কাজ করে কি না? না করলে কেন করবে না?

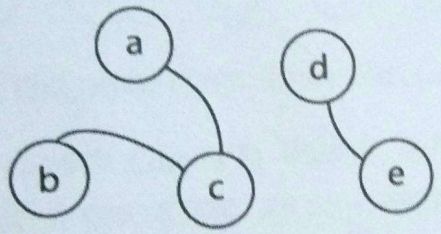
Scanned by CamScanner

অধ্যায় ৬ – ডিসজয়েন্ট সেট (Disjoint Set)

ডিসজয়েন্ট সেট (Disjoint set) সাধারণত ডেটা স্ট্রাকচার টপিকের মধ্যে পড়ানো হয়। কিন্তু আমরা দুটি কারণে এখন এটি শিখব। এটি গ্রাফ সম্পর্কিত একটি ডেটা স্ট্রাকচার এবং গ্রাফ অ্যালগরিদম শিখতে এবং প্রবলেম সলভ করতে এটি আমাদের অনেক কাজে লাগবে। তৃতীয় আরেকটি কারণ হিসেবে বলা যায় এটি শেখা খুব সহজ!

প্রথমে আমরা দেখব কি ধরনের কাজে আমাদের এই ডেটা স্ট্রাকচারটি দরকার। মনে করি A, B, C, D ও E পাঁচজন মানুষ। $[A - B]$ যদি পরস্পরের বন্ধু হয় এবং $[B - C]$ যদি পরস্পরের বন্ধু হয় তাহলে আমরা বলতে পারি $[A - C]$ ও পরস্পরের বন্ধু, অর্থাৎ তাদের বন্ধুত্ব ট্রানজিটিভ (transitive)।

এখন আমি বলে দিলাম যে $[A - B], [B - C]$এবং $[D - E]$ এরা পরস্পরের বন্ধু। এরপর তোমাকে প্রশ্ন করলাম $[A - C]$ কি পরস্পরের বন্ধু? $[B - D]$ কি পরস্পরের বন্ধু? প্রথম প্রশ্নের উত্তর হবে "হ্যা", দ্বিতীয় প্রশ্নের উত্তর হবে "না"। আমাদের তথ্যগুলোকে গ্রাফের মাধ্যমে দেখাতে পারি (চিত্র ৬.১)।



চিত্র ৬.১: পাঁচজন মানুষের মধ্যে বন্ধুত্বের গ্রাফ

সহজেই বোঝা যাচ্ছে গ্রাফে যেসব নোড একই কম্পোনেন্টের মধ্যে আছে তারা পরস্পরের বন্ধু। তাহলে দুজন ব্যক্তি পরস্পরের বন্ধু কি না সেটি জানতে হলে আমাদের দেখতে হবে তারা একই কম্পোনেন্টে আছে কি না। সহজেই বিএফএস বা ডিএফএস অ্যালগরিদম চালিয়ে এটি বের করা যায়। কিন্তু এগুলোর থেকেও ভালো উপায় হলো "ডিসজয়েন্ট সেট", কেন এটি ভালো সেটি এখনই বুঝতে পারবে।

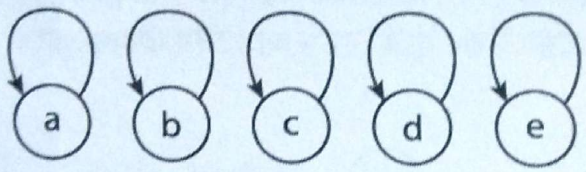
ডিসজয়েন্ট সেট কাকে বলে? একাধিক সেট, যাদের মধ্যে কোনো সাধারন উপাদান (Common Element) নেই, তাদের ডিসজয়েন্ট সেট বলা হয়। যেমন: $A = \{1\}, B = \{2, 5, 7\}, C = \{3, 6\}$ এই তিনটি সেট ডিসজয়েন্ট।

Scanned by CamScanner

আমাদের লক্ষ হবে গ্রাফের নোডগুলোকে এক বা একাধিক ডিসজয়েন্ট সেটে ভাগ করে ফেল দুটি নোড একই কম্পোনেন্টে থাকলে তারা একই সেটে থাকবে। তার মানে, দুটি নোড এ কম্পোনেন্টে আছে কি না জানতে হলে আমাদের দেখতে হবে নোড দুটি একই সেটে আছে না।

প্রতিটি সেটের একটি করে রিপ্রেজেন্টেটিভ (Representative) থাকবে। রিপ্রেজেন্টেটিভ এখানে তুমি ক্রিকেট দলের মধ্যে ক্যাপ্টেনের মতো করে ভাবতে পারো। সাকিব আর তা একই দলে খেলে কি না জানতে তুমি দেখতো পার তাদের দলের ক্যাপ্টেন কে? দুজনেরই দ ক্যাপ্টেন মাশরাফি, তারমানে তারা একই দলে খেলে বা গ্রাফের একই কম্পোনেন্টের সদস কিন্তু সাকিব আর ক্রিস গেইল ভিন্ন ভিন্ন ক্যাপ্টেনের অধীনে খেলে, তার মানে তারা গ্রাফের এ কম্পোনেন্টে নেই।

চিত্রসহ উদাহরণ দেখলে ব্যাপারটি সহজ হয়ে যাবে। শুরুতে কেউ কারও বন্ধু নয়। তাহলে সব আলাদা আলাদা সেটে আছে এবং যেহেতু সেটগুলোতে মাত্র একটি করে সদস্য তাই সে সদস্যটিই হবে সেটের রিপ্রেজেন্টেটিভ (চিত্র ৬.২)।



চিত্র ৬.২: শুরুতে সবাই আলাদা আলাদা সেটে আছে

সেটে প্রতিটি নোডের একটি করে প্যারেন্ট (Parent) থাকবে। $parent[u]$ দিয়ে বোঝা নোড u সরাসরি কার সঙ্গে সংযুক্ত। একদম শুরুতে ধরে নিচ্ছি প্রতিটি নোডের প্যারেন্ট নিজেই।

এই অংশটির সুডোকোড হবে এরকম,

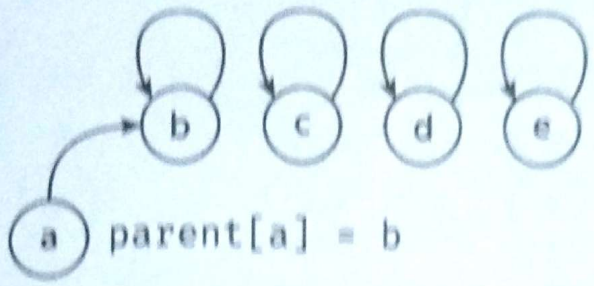
```
procedure makeset(u):
  parent[u] ← u
  return

procedure init_disjoint_set(N):
  for i from 1 to N do
    makeset(i)
  return
```

সুডোকোড ৬.১

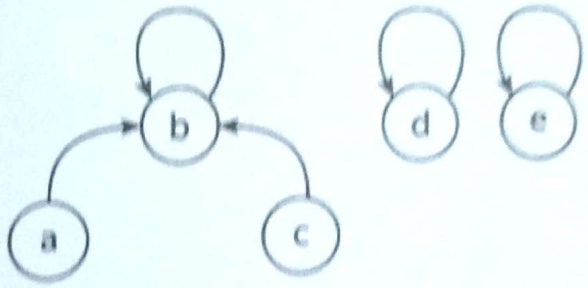
Scanned by CamScanner

এখন a এবং b হঠাৎ ঠিক করল যে তারা বন্ধু হবে। তাহলে তাদের একই সেটের মধ্যে আসতে হবে। a ঠিক করল যে b এর রিপ্রেজেন্টেটিভকে নিজের রিপ্রেজেন্টেটিভ বানিয়ে ফেলবে, তাহলেই তারা একই সেটে চলে আসবে। তাই a এসে b এর সঙ্গে যুক্ত হয়ে গেল কারণ, b এর রিপ্রেজেন্টেটিভ হলো b নিজেই। চিত্র ৬.৩ দেখো।



চিত্র ৬.৩: a এর রিপ্রেজেন্টেটিভ এবং প্যারেন্ট এখন b

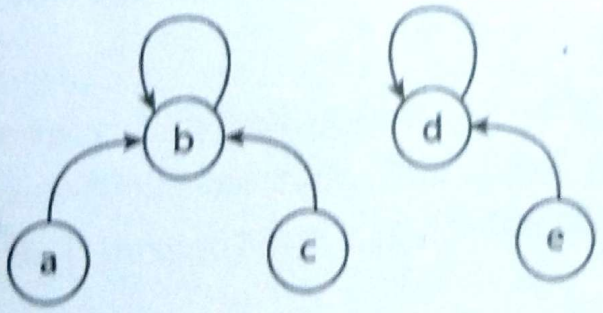
এখন c ভাবল সেও a এর বন্ধু হবে। এখন c কার নিচে যুক্ত হবে? দুজনকে একই দলে আসতে হলে তাদের ক্যাপ্টেন বা রিপ্রেজেন্টেটিভ অভিন্ন হতে হবে, c তাই যুক্ত হবে a এর রিপ্রেজেন্টেটিভের নিচে, তার মানে b এর নিচে (চিত্র ৬.৪),



parent[c] = representative(a) = b

চিত্র ৬.৪: c এর রিপ্রেজেন্টেটিভ এবং প্যারেন্ট এখন b

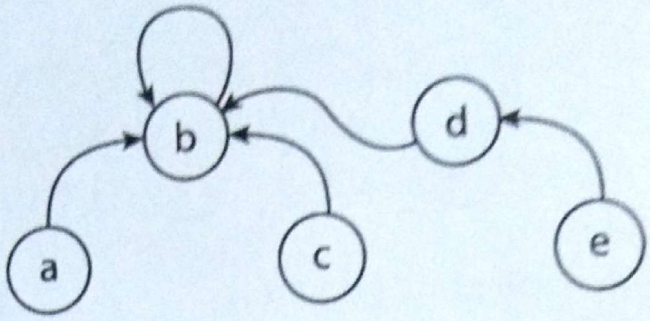
d, e কেও এই ফাঁকে বন্ধু বানিয়ে দেই (চিত্র ৬.৫),



চিত্র ৬.৫: e এর রিপ্রেজেন্টেটিভ এবং প্যারেন্ট এখন d

Scanned by CamScanner

এবার আমরা e এবং c কে বন্ধু বানাব। c যেই সেটে সেটির রিপ্রেজেন্টেটিভ হলো b। তার প
আমরা যুক্ত করবো e এর রিপ্রেজেন্টেটিভকে। e এর রিপ্রেজেন্টেটিভ হলো d। b কে d
প্যারেন্ট বানিয়ে দিলাম। তাহলে b এখন e এরও রিপ্রেজেন্টেটিভ হয়ে গেল (চিত্র ৬.৬)।



চিত্র ৬.৬: d এর রিপ্রেজেন্টেটিভ এবং প্যারেন্ট এখন b

আমরা আবার একটু সুডোকোড দেখি:

```
procedure Union(a, b):
  u = FindRepresentative(a)
  v = FindRepresentative(b)

  if u != v
    parent[u] ← v
  end if

  return
```

সুডোকোড ৬.২

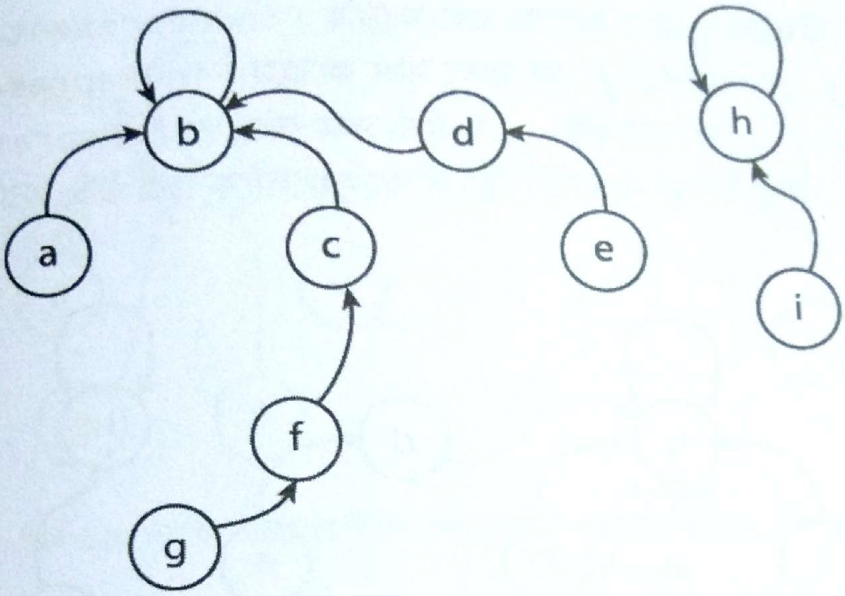
আমরা যে দুটি নোডকে বন্ধু বানাব তাদের রিপ্রেজেন্টেটিভ বের করল
`FindRepresentative()` ফাংশন কল করে (ফাংশনটির সংজ্ঞা পরে দেখবো)। এব
একটিকে আরেকটির প্যারেন্ট বানিয়ে দিলাম।

এখন প্রশ্ন হলো, রিপ্রেজেন্টেটিভ খুঁজব কীভাবে? মনোযোগ দিয়ে এই পর্যন্ত পড়ে থাকলে এব
জিনিস চোখে পড়ার কথা, কোনো একটি সেটের রিপ্রেজেন্টেটিভ এলিমেন্টের প্যারেন্ট
এলিমেন্ট নিজেই, মানে এলিমেন্টটি যদি হয় r তাহলে $parent[r] = r$।

Scanned by CamScanner

কোনো একটি সেটের রিপ্রেজেন্টেটিভ এলিমেন্টের প্যারেন্ট সেই এলিমেন্ট নিজেই, অথবা বলা যায় যে নোডের প্যারেন্ট সে নিজেই সেই নোডটিই ওই সেটের রিপ্রেজেন্টেটিভ। এলিমেন্টটি যদি হয় r তাহলে $parent[r] = r$।

আমরা আরেকটু বড় গ্রাফ দেখি। মনে করি a, b, c, d ও e এর বন্ধু হতে আরও কয়েকজন চলে এসেছে (চিত্র ৬.৭)।



চিত্র ৬.৭: আরো কিছু নতুন বন্ধু গ্রাফে যুক্ত হয়েছে

আগের সেটেই শুধু কিছু নোড যুক্ত করা হয়েছে পরের অংশ বোঝার সুবিধার জন্য। ধরি f এর রিপ্রেজেন্টেটিভ বের করতে হবে। f এর প্যারেন্ট সে নিজে নয়, তাহলে তার রিপ্রেজেন্টেটিভ খুঁজতে তার প্যারেন্টের প্যারেন্ট চেক করি। f এর প্যারেন্ট হচ্ছে c। c ও রিপ্রেজেন্টেটিভ নয় কারণ সে নিজেই নিজের প্যারেন্ট নয়, c এর প্যারেন্ট আবার b এবং b নিজেই নিজের প্যারেন্ট। তাহলে b ই হলো f এর রিপ্রেজেন্টেটিভ। ফাংশনটি লিখে ফেলি,

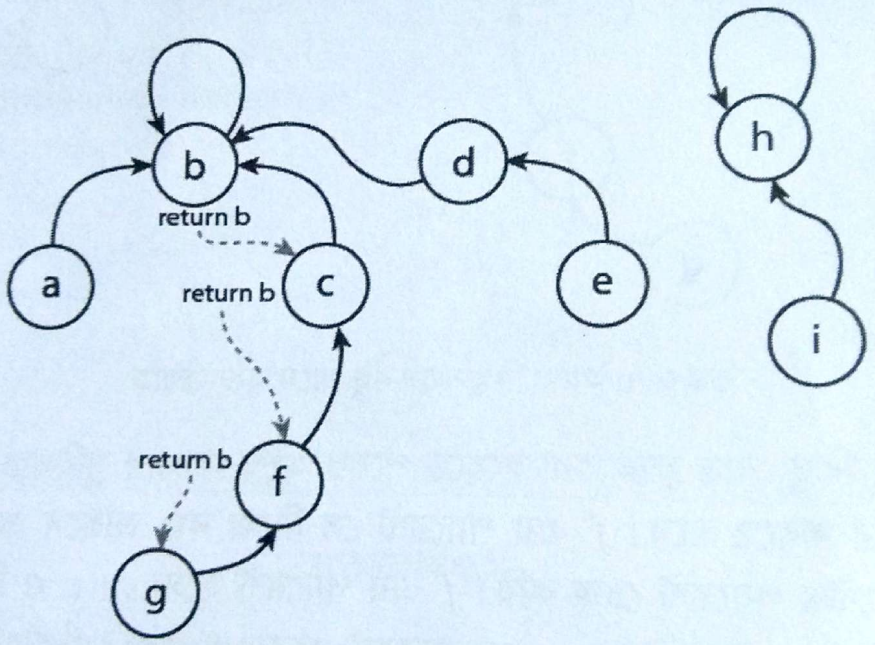
```
procedure FindRepresentative(r):  #Without path compression
  if parent[r] = r
    return r
  end if
  return FindRepresentative(parent[r])
```

সুডোকোড ৬.৩

Scanned by CamScanner

ফাংশনটি ট্রি বেয়ে উপরে উঠল যতক্ষন না রিপ্রেজেন্টেটিভকে খুঁজে পাওয়া যায়। আগের ফাংশ
দুটি (union এবং makeset) তাদের আসল কাজ করেছে $O(1)$ ও
FindRepresentative() ফাংশনে এসে কমপ্লেক্সিটির প্রশ্ন এসে পড়েছে। worst case
$O(n)$ সময় লাগতে পারে উপরের FindRepresentative() ফাংশনের, ট্রিয়ের গভীর
বা ডেপথ (Depth) যত বেশি হবে তত সময় বেশি লাগবে।

এখানে একটি চমৎকার অপটিমাইজেশন আছে। প্রতিটি FindRepresentative() ফাংশ
কলের সঙ্গে সঙ্গে আমরা ট্রিটিকে এমন ভাবে পরিবর্তন করব যেন গভীরতা (depth) কমে যায়
g এর রিপ্রেজেন্টেটিভকে খুঁজতে আমরা কল করেছি FindRepresentative(g)। এ
ফাংশনটি আবার g এর প্যারেন্ট f এর জন্য কল করেছে FindRepresentative(f)
সেই ফাংশনটি আবার f এর প্যারেন্ট c এর জন্য কল করে FindRepresentative(c
রিপ্রেজেন্টেটিভকে খুঁজে রিটার্ন করেছে। রিটার্ন ভ্যালুগুলোকে চিত্র ৬.৮ এর মত করে দেখা
পারি।



চিত্র ৬.৮: FindRepresentative(g) ফাংশনের রিটার্ন ভ্যালু

চিত্রে প্রতিটি নোডের পাশে রিটার্ন ভ্যালু লিখে দেওয়া হয়েছে। প্রতিটি নোড তার আগের ফাংশ
একই ভ্যালু রিটার্ন করে, যেই ভ্যালুটি আমরা সবশেষে পাই, সেই ভ্যালুটিই রিপ্রেজেন্টেটিভ।

এখন আমরা যদি সুডোকোডটি এভাবে লিখতাম তাহলে কি ঘটতো?

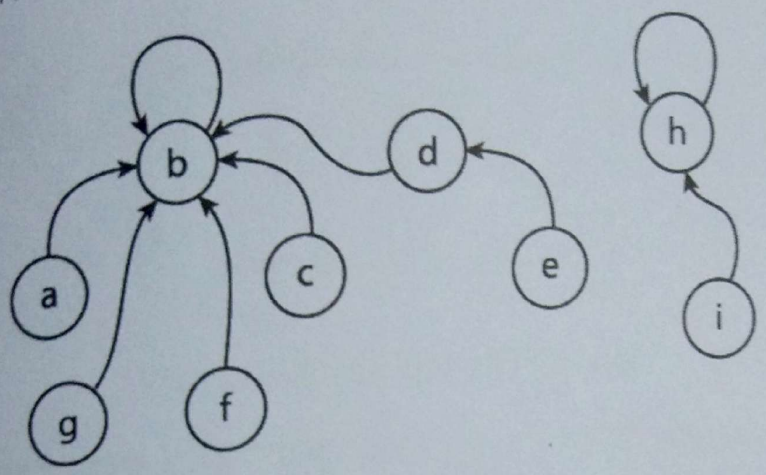
```
procedure FindRepresentative(r):
  if parent[r] = r
    return r
  end if
```

Scanned by CamScanner

```
    parent[r] ← FindRepresentative(parent[r])
    return parent[r]
```

সুডোকোড ৬.৪

এটি করলে রিটার্ন করার সময় রিটার্ন ভ্যালুটিকে নিজের প্যারেন্ট বানিয়ে ফেলতো, অর্থাৎ সবাই রিপ্রেজেন্টেটিভকেই বানিয়ে ফেলত নিজের প্যারেন্ট! ফাংশন কলের সঙ্গে সঙ্গে গ্রাফটি হয়ে যেত চিত্র ৬.৯ এর মতো।



চিত্র ৬.৯: সবাই রিপ্রেজেন্টেটিভকেই নিজের প্যারেন্ট বানিয়ে ফেলেছে

ট্রিয়ের গভীরতা কমে গিয়েছে, তাই নয় কি? একে বলা হয় পাথ কমপ্রেশন (Path Compression)। পরে আমরা আবার যখন f এর রিপ্রেজেন্টেটিভ খুঁজব তখন আর মাত্র একটি এজ বেয়ে উঠতে হবে।

পাথ কমপ্রেশন ব্যবহার করে m সংখ্যকবার `FindRepresentative` অপারেশন চালালে টাইম কমপ্লেক্সিটি হবে $O(m \log n)$। তুমি যদি আগ্রহী হও তাহলে ইন্টারনেট বা ভালো কোনো বই থেকে এ কথার প্রমাণটি জেনে নিতে পারো।

পাথ কমপ্রেশন ব্যবহার করে m সংখ্যকবার `FindRepresentative` অপারেশন চালালে টাইম কমপ্লেক্সিটি হবে $O(m \log n)$।

ডিসজয়েন্ট সেট ব্যবহার করে আমরা ক্রুসকাল (Kruskal) অ্যালগরিদম ইমপ্লিমেন্টেশন শিখব পরের অধ্যায়ে।

Scanned by CamScanner

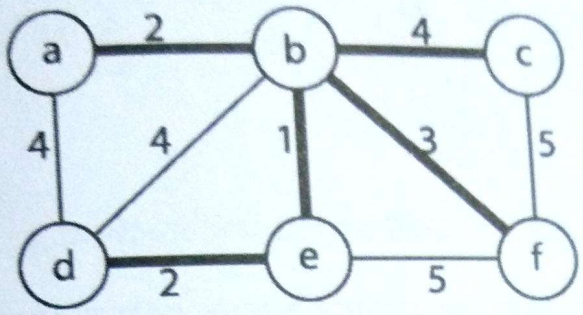
চিন্তা করার মত সমস্যা

ডিসজয়েন্টে সেটে প্রতিটি সেটের সাইজ কিভাবে বের করা যেতে পারে?

Scanned by CamScanner

অধ্যায় ৭ – মিনিমাম স্প্যানিং ট্রি (Minimum Spanning Tree)

এখন আমরা দুটি অ্যালগরিদম শিখব মিনিমাম স্প্যানিং ট্রি (Minimum Spanning Tree) সম্পর্কিত। প্রথমেই আমরা জেনে নেই মিনিমাম স্প্যানিং ট্রি জিনিসটি কী। সে জন্য চিত্র ৭.১ দেখ।



চিত্র ৭.১: একটি ওয়েটেড আনডিরেক্টেড গ্রাফ

আমরা একটি ওয়েটেড গ্রাফ দেখতে পাচ্ছি। এখন মনে কর, প্রতিটি নোড হল একটি করে বাড়ি। এদের মধ্যে টেলিফোনের তার বসানো দরকার যেন সব বাড়ি থেকে সব বাড়িতে ফোন করা যায় ফোন করার জন্য সরাসরি এজ থাকার দরকার নেই, অন্যান্য বাড়ি ঘুরে একটা পাথ পাওয়া গেলেই চলবে। ছবিতে গাঢ় এজ দিয়ে তার বসানোর একটি উপায় দেখানো হয়েছে। বাকি এজগুলো বাদ দিলে যে সাবগ্রাফটি পেলাম সেটি মূল গ্রাফের একটি "স্প্যানিং ট্রি (Spanning Tree)"। এই স্প্যানিং ট্রি তৈরির মোট কস্ট (cost) হলো এজ গুলোর ওয়েটের যোগফল, এক্ষেত্রে $1 + 2 + 2 + 3 + 4 = 12$।

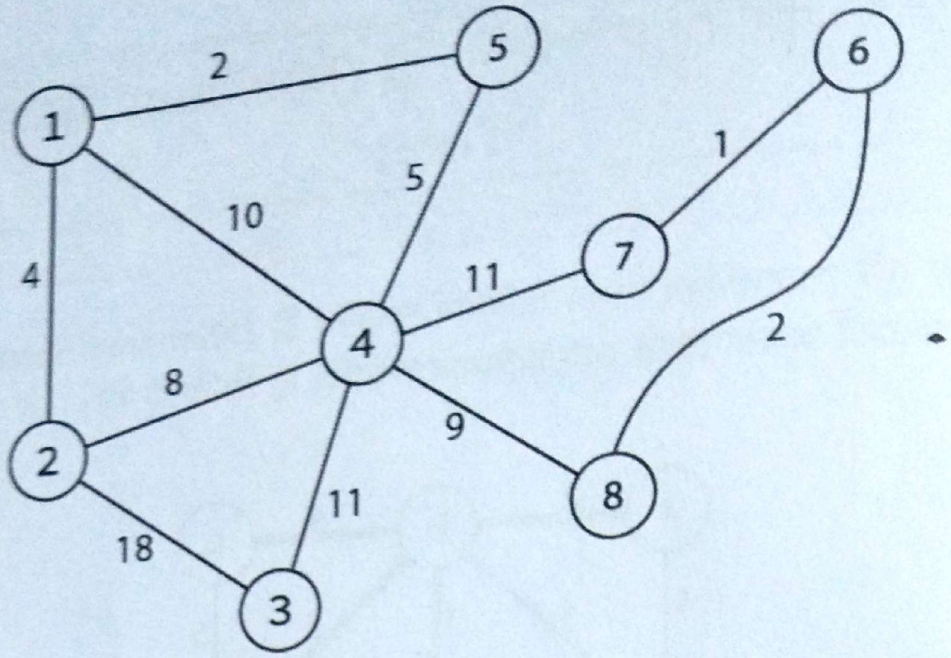
একটি গ্রাফের একাধিক স্প্যানিং ট্রি থাকতে পারে। যে স্প্যানিং ট্রিয়ের কস্ট সব থেকে কম সেটিই মিনিমাম স্প্যানিং ট্রি।

মিনিমাম স্প্যানিং ট্রি বের করার দুটি অ্যালগরিদম আমরা শিখব।

৭.১ – প্রিম অ্যালগরিদম (Prim's Algorithm)

চিত্র ৭.২ এর গ্রাফটি থেকে আমরা রবার্ট প্রিম (Robert C. Prim) এর আবিষ্কার করা অ্যালগরিদম ব্যবহার করে মিনিমাম স্প্যানিং ট্রি বের করব। সহজে বোঝার জন্য শুরুতে আমরা একটি সিমুলেশন দেখি।

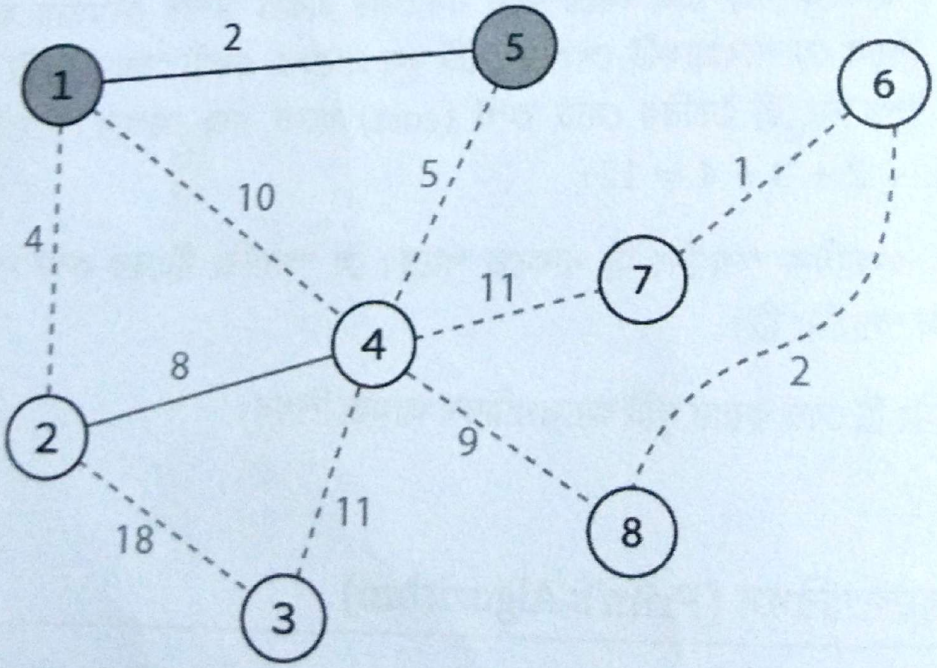
Scanned by CamScanner



চিত্র ৭.২: একটি ওয়েটেড আনডিরেক্টেড গ্রাফ

S_t নামের একটি সাবগ্রাফ কল্পনা কর যেখানে শুরুতে কোনো নোড বা এজ নেই। আমরা গ্রাফ থেকে একটি একটি এজ এই সাবগ্রাফে যোগ করে মিনিমাম স্প্যানিং ট্রি গঠন করব।

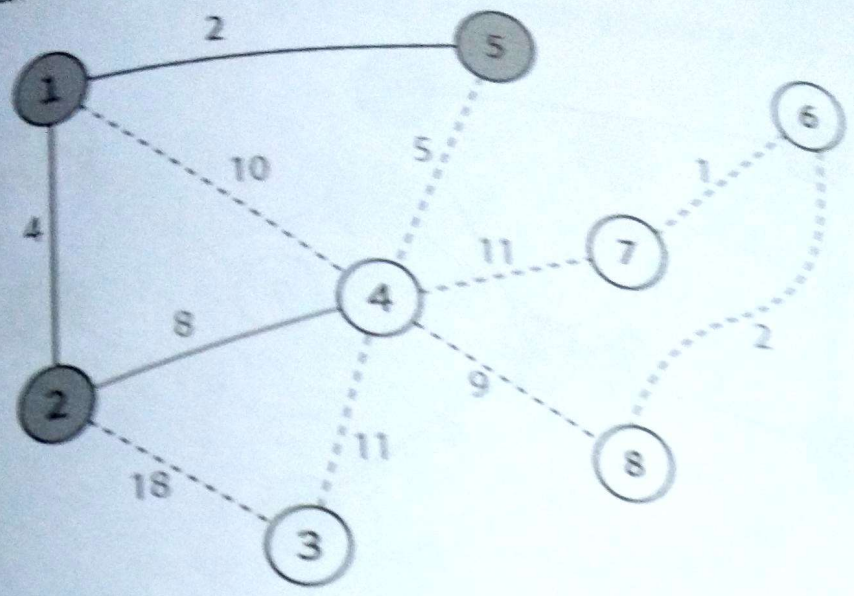
আমরা প্রথমে যেকোনো একটি সোর্স নোড নিব। ধরি সোর্স নোড হলো 1। নোড 1 থে যতগুলো এজ আছে সেগুলোর মধ্যে মিনিমাম ওয়েটের এজটিকে আমরা সাবগ্রাফে যোগ ক চিত্র ৭.৩ এ সলিড এজ দিয়ে বোঝানো হচ্ছে একটি সাবগ্রাফে যুক্ত করা হয়েছে।



চিত্র ৭.৩: [1 – 5] এজটিকে সাবগ্রাফে যুক্ত করা হয়েছে

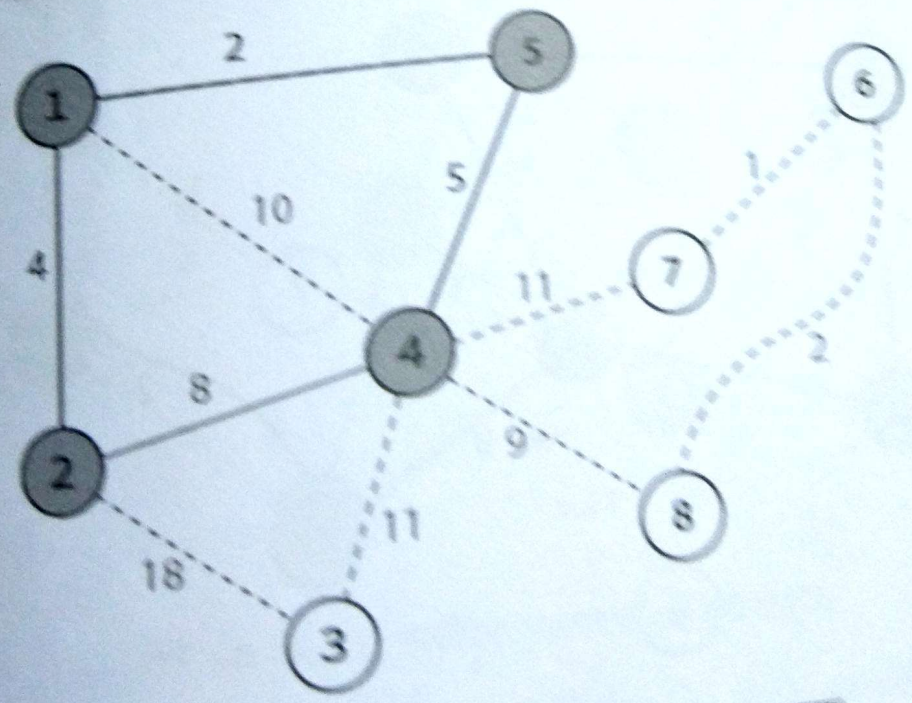
Scanned by CamScanner

এবার সোর্স নোড 1 এবং নোড 5 থেকে মোট যত এজ আছে তাদের মধ্যে মিনিমাম ওয়েটের একটিকে সাবগ্রাফে যুক্ত করব (চিত্র ৭.৪)।



চিত্র ৭.৪: [1 – 2] একটিকে সাবগ্রাফে যুক্ত করা হয়েছে

এবার নোড 1, নোড 2 এবং নোড 5 থেকে মোট যত এজ আছে তাদের মধ্যে মিনিমাম ওয়েটের একটিকে সাবগ্রাফে যোগ করব (চিত্র ৭.৫)।

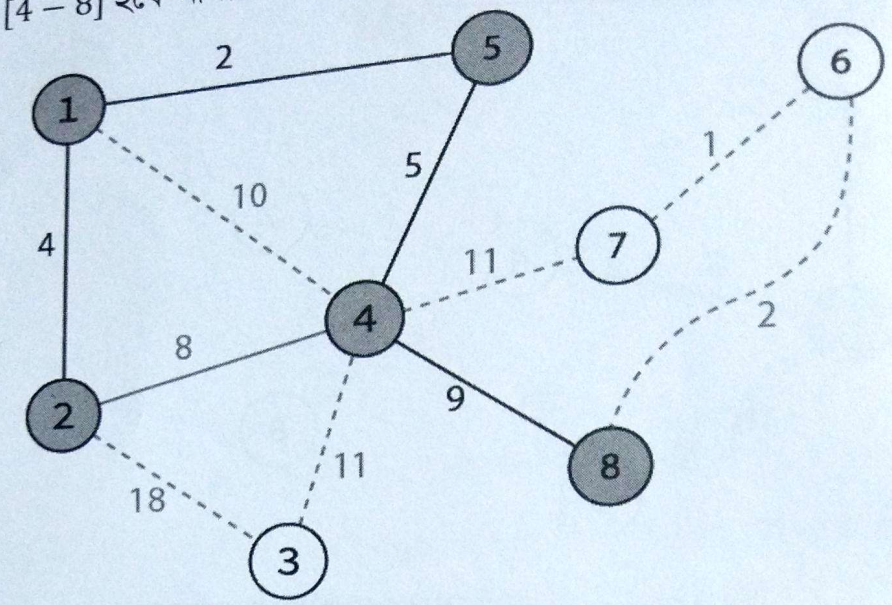


চিত্র ৭.৫: [5 – 4] একটিকে সাবগ্রাফে যুক্ত করা হয়েছে

পরের ধাপটি গুরুত্বপূর্ণ (চিত্র ৭.৬)। নোড 1, নোড 2, নোড 5 এবং নোড 4 থেকে যত এজ আছে তাদের মধ্য মিনিমাম হলো [2 – 4], কিন্তু নোড 2 এবং নোড 4 দুটিই ইতিমধ্যে সাবগ্রাফের

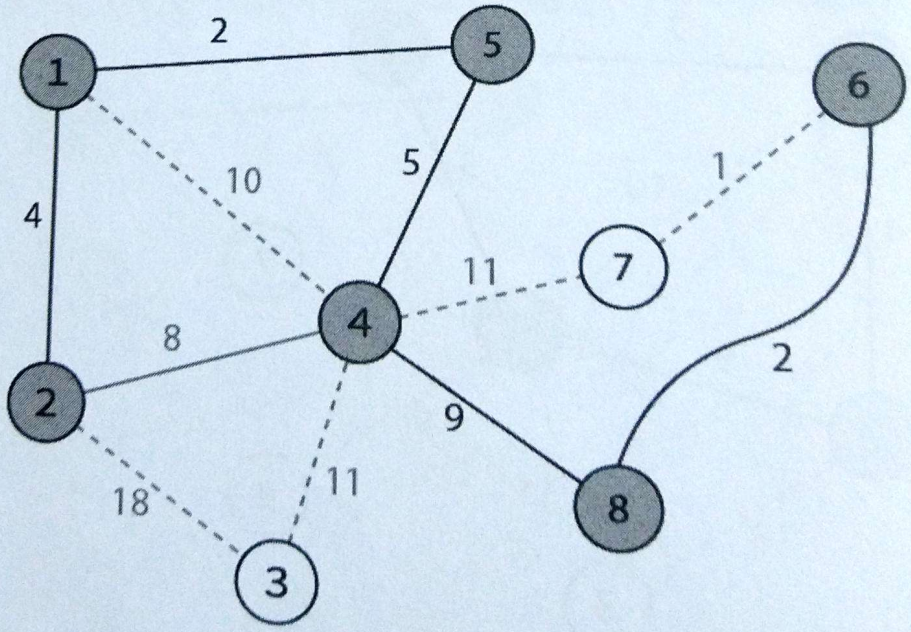
Scanned by CamScanner

অংশ, এদের যোগ করলে সাবগ্রাফে সাইকেল তৈরি হবে, তাই [2 – 4] এজটি নিয়ে আমাদের কোনো লাভ হবে না। আমরা প্রতিবার এমন এজ নিব যেন নতুন আরেকটি নোড সাবগ্রাফে যুক্ত হয়। তাহলে [4 – 8] হবে আমাদের পরের পছন্দ (চিত্র ৭.৬)।



চিত্র ৭.৬: [4 – 8] এজটিকে সাবগ্রাফে যুক্ত করা হয়েছে

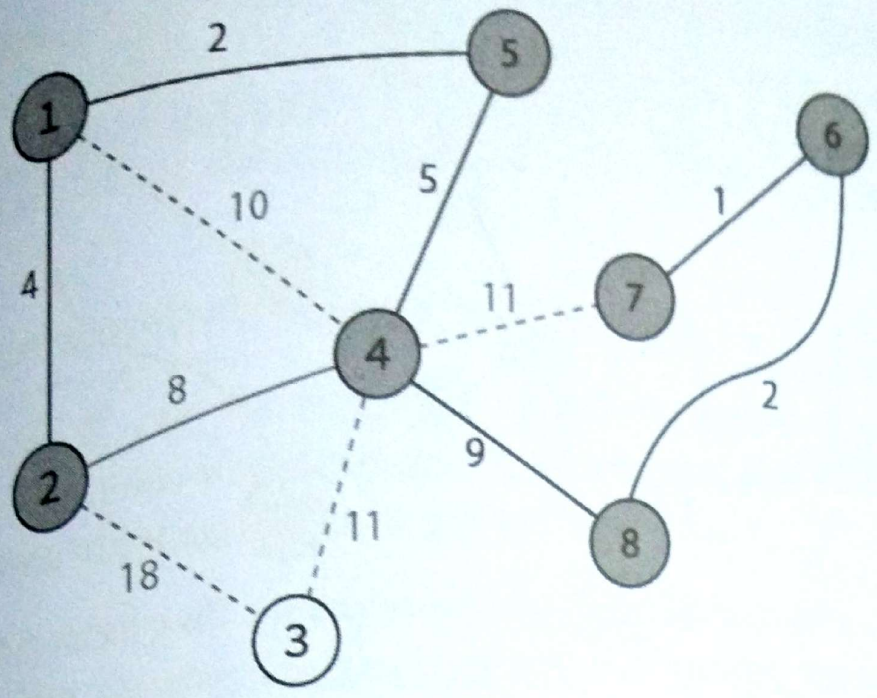
এরপর [8 – 6] এজটিকে যুক্ত করব (চিত্র ৭.৭)।



চিত্র ৭.৭: [8 – 6] এজটিকে সাবগ্রাফে যুক্ত করা হয়েছে

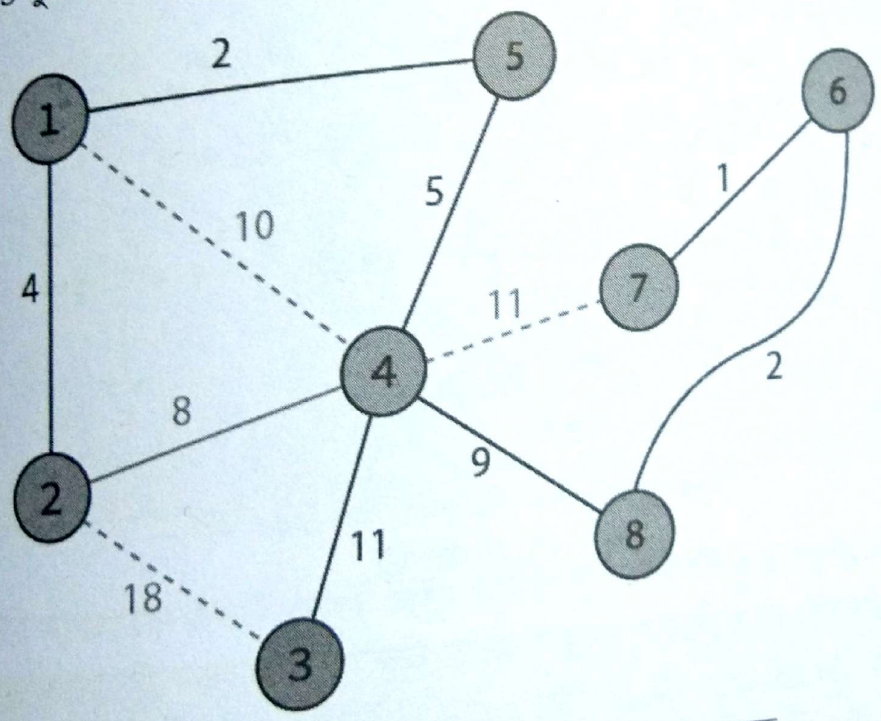
এরপর যুক্ত হবে [6 – 7] (চিত্র ৭.৮)।

Scanned by CamScanner



চিত্র ৭.৮: [6 – 7] এজটিকে সাবগ্রাফে যুক্ত করা হয়েছে

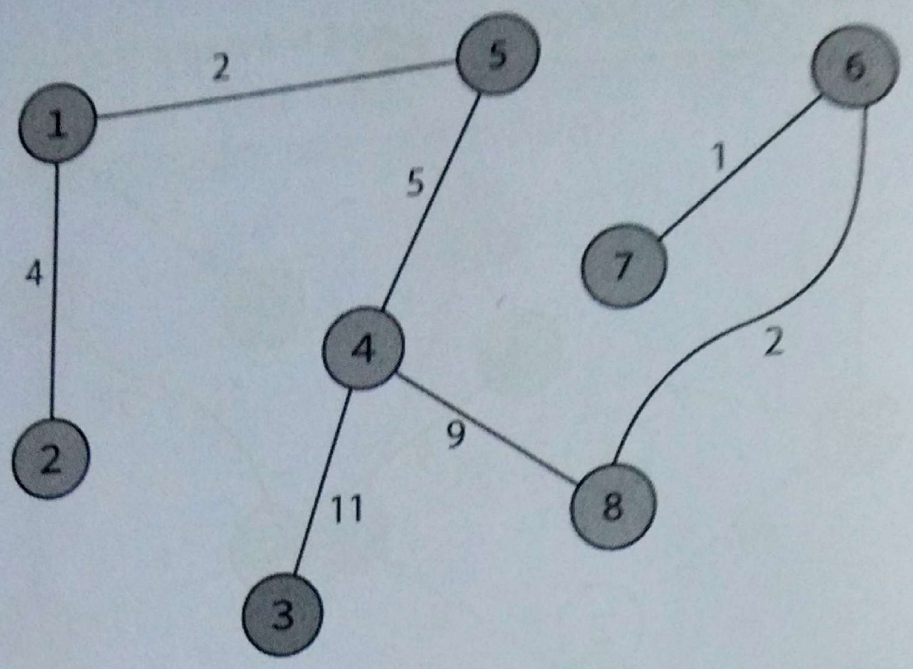
সবশেষে 4 – 3 যুক্ত করি (চিত্র ৭.৯)।



চিত্র ৭.৯: [4 – 3] এজটিকে সাবগ্রাফে যুক্ত করা হয়েছে

গাঢ় রঙের এই সাবগ্রাফটিই আমাদের মিনিমাম স্প্যানিং ট্রি। বাকি এজগুলো মুছে দিলে চিত্র ৭.১০ এর মতো একটি গ্রাফ থাকে।

Scanned by CamScanner



চিত্র ৭.১০: চিত্র ৭.২ এর গ্রাফের মিনিমাম স্প্যানিং ট্রি

সবগুলো এজের ওয়েটের যোগফল: $4 + 2 + 5 + 11 + 9 + 2 + 1 = 34$।

```
procedure PrimsMST(G):
  Vt={x} where x is an arbitrary node (starting point)
  from V.
  St={}
  while Vt != G.V
    Choose an edge (u, v) with minimal weight such that u
    is in Vt and v is not
    Add v to Vt, and (u, v) to St
  end while
  Return St
```

সুডোকোড ৭.১

প্রোগ্রামিং টিপস

সুডোকোডের 4 নম্বর ধাপে মিনিমাম ওয়েটের এজ খুঁজতে বারবার সবগুলো এজের উপর লু
চালানো হলে কমপ্লেক্সিটি অনেক বেশি হয়ে যাবে $O(VE)$। রানটাইম কমাতে প্রায়োরিটি কি
ব্যবহার করতে পার। যখন নতুন একটি নোড V_t তে যোগ করছ তখন সেই নোডে
অ্যাডজেসেন্ট সবগুলো এজ প্রায়োরিটি কিউতে ঢুকিয়ে রাখতে হবে। 4 নম্বর ধাপে প্রায়োরি

Scanned by CamScanner

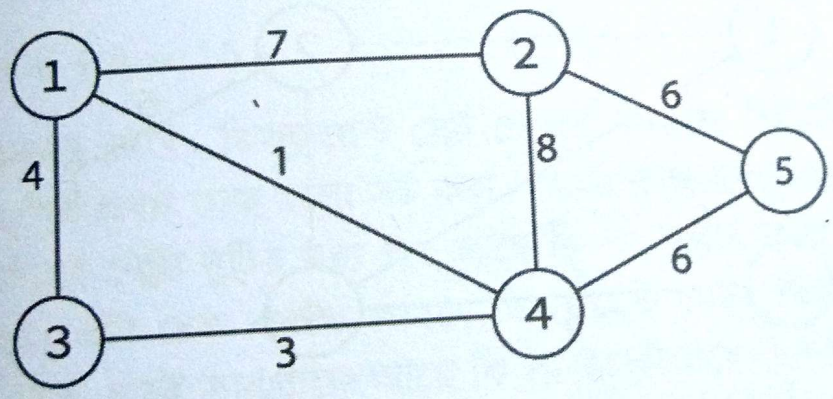
কিউ থেকে সবচেয়ে মিনিমাম ওয়েটের এজটি লগারিদমিক কমপ্লেক্সিটিতে খুঁজতে পারবে। মোট কমপ্লেক্সিটি হবে $O(E \log E)$। কমপ্লেক্সিটি $O(E \log V)$ তেও নামিয়ে আনা সম্ভব, সেটি কীভাবে করা যায় চিন্তা করে বের কর।

৭.২ – ক্রুসকাল অ্যালগরিদম (Kruskal's Algorithm)

মিনিমাম স্প্যানিং ট্রি বের করার আরেকটি অ্যালগরিদম হলো ক্রুসকাল অ্যালগরিদম (Kruskal's Algorithm)। আমরা ডিসজয়েন্ট সেট ডেটা স্ট্রাকচার কীভাবে কাজ করে শিখেছি, এখন সেটি ব্যবহার করে আমরা ক্রুসকাল কীভাবে কাজ করে শিখব।

আগের মতোই S_t নামের একটি সাবগ্রাফ কল্পনা কর যেখানে শুরুতে কোনো নোড বা এজ নেই। ক্রুসকাল অ্যালগরিদমেও আমরা একটি করে এজ নিয়ে সাবগ্রাফে যোগ করে মিনিমাম স্প্যানিং ট্রি তৈরি করার চেষ্টা করব। এবার আমরা এজগুলো নিব ওয়েট বা কস্ট অনুযায়ী, ছোট থেকে বড়। যদি নতুন এজটি একটি সাইকেল তৈরি করে তাহলে সেটি নিব না, যদি না করে তাহলে তাকে সাবগ্রাফে যোগ করব।

আমরা চিত্র ৭.১১ এর গ্রাফে একটি সিমুলেশন করে ফেলি।



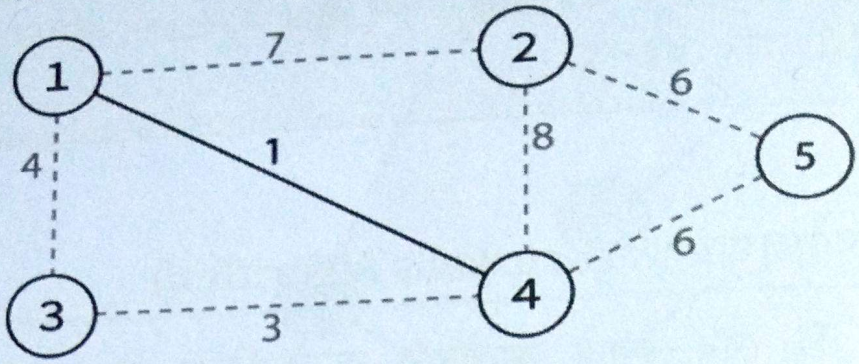
চিত্র ৭.১১: একটি ওয়েটেড আনডিরেক্টেড গ্রাফ

প্রথমে আমাদের সাবগ্রাফে একটি এজও নেই। আমরা মূল গ্রাফের এজগুলোকে ওয়েট অনুযায়ী ছোট থেকে বড় ক্রমে সর্ট করে ফেলব। দুটি এজের ওয়েট সমান হলে যেকোনো একটি আগে নিতে পারি। তারপর একটি করে এজ নিব আর দেখব এজের দুই প্রান্তের নোডগুলোর মধ্যে ইতিমধ্যে কোনো পথ আছে কি না, যদি থাকে তাহলে এজটি নিলে সাইকেল তৈরি হবে, তাই এজটি আমরা নিবনা।

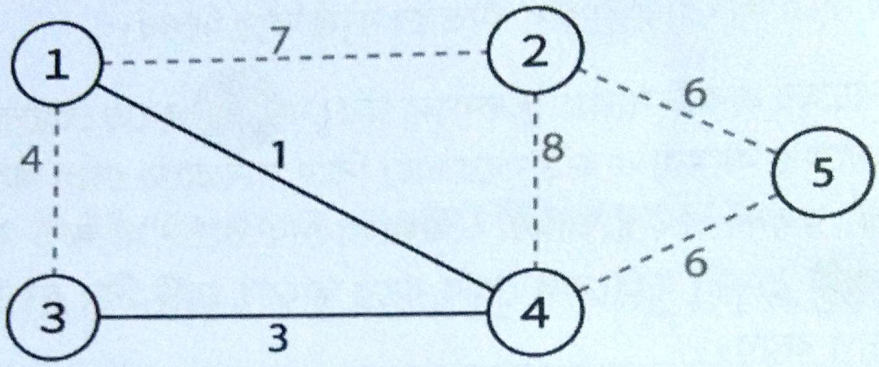
চিত্র ৭.১১ তে $[1-4]$ হলো সবথেকে কম কস্টের এজ, এই এজটিকে সাবগ্রাফে যুক্ত করি (চিত্র ৭.১২)।

Scanned by CamScanner

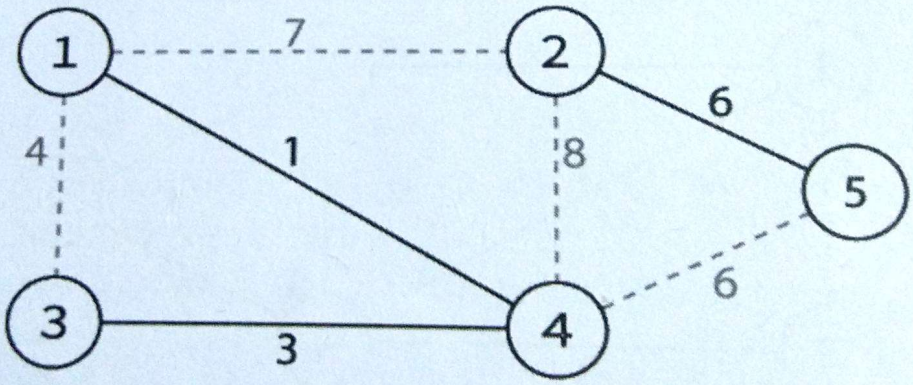
এরপর [3 – 4] এজটিকে সাবগ্রাফে যোগ করব (চিত্র ৭.১৩)।



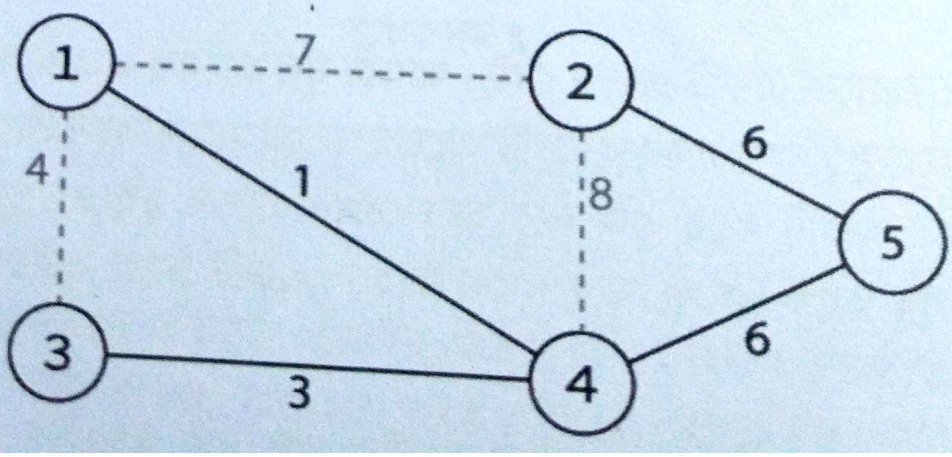
চিত্র ৭.১২: [1 – 4] এজটিকে সাবগ্রাফে যুক্ত করা হয়েছে



চিত্র ৭.১৩: [3 – 4] এজটিকে সাবগ্রাফে যুক্ত করা হয়েছে



চিত্র ৭.১৪: [2 – 5] এজটিকে সাবগ্রাফে যুক্ত করা হয়েছে



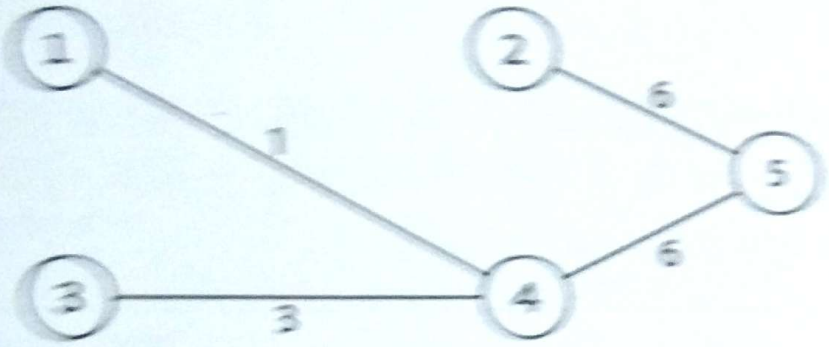
Scanned by CamScanner

এখন সবচেয়ে কম কস্টের এজ হলো [1 – 3] যার কস্ট 4। কিন্তু এই এজটি যুক্ত করলে সাইকেল তৈরি হবে, এদিকে আমরা বাদ দিতে দিতে পারি।

এখন [2 – 5] বা [4 – 5] এই দুটি এজের যেকোনো একটিকে নিতে পারি কারণ এদের কস্ট একই। আমি [2 – 5] কে নিব (চিত্র ৭.১৪)।

এখন নিব [4 – 5] কে (চিত্র ৭.১৫)।

[1 – 2] এবং [2 – 4] কে কোনোভাবে লাগবার নেই, কারণ তাতে সাইকেল তৈরি হবে। মোটা রেখাতে আঁকা এজের সাবগ্রাফটিই আমাদের মিনিমাম স্প্যানিং ট্রি, বাকি এজগুলো মুছে দিলে চিত্র ৭.১৬ এর মতো ট্রি পাওয়া যায়।



চিত্র ৭.১৬: চিত্র ৭.১১ এর গ্রাফের মিনিমাম স্প্যানিং ট্রি

মোট কস্ট: $1 + 3 + 6 + 6 = 16$।

এখন আমরা সুডোকোড দেখি। ডিসজয়েন্ট সেট শেখার সময় যা শিখেছিলাম সেসব এখানে কাজে লাগবে। আমাদের প্রথম কাজ হলো সর্ট করা। পরের কাজ হলো একটি একটি এজ নিয়ে চেক করা যে দুই প্রান্তের নোড দুটির মধ্যে পথ আছে কি না, অর্থাৎ তারা একই কম্পোনেন্টের ভেতরে আছে কি না। এটা চেক করতে লাগবে ডিসজয়েন্ট সেট। ডিসজয়েন্ট সেট অধ্যায়ে শিখেছিলাম দুটি নোড একই সেটগ্রাফে আছে কি না তা কীভাবে বের করতে হয়। তুমি সেই কাজটিই এখানে করবে। তারপর একই সেটগ্রাফে না থাকলে আগের মতো ইউনিয়ন ফাংশন কল করে ফাংশনের এজটিকে নিয়ে আসবে আর একটি সেট করে রাখবে।

```
procedure kruskalMST(G):
    Sort E
    E_sorted = sort_by_weight(G,E)
    parent[N] = NULL

    for i from 1 to N do
        makeset(N)
        for all the edges (u,v) in E_sorted do
```

Scanned by CamScanner

```
        if FindRepresentative(u) != FindRepresentative(v)
          Union(u,v)
            add (u,v) to St
        end if
      end for
    return St
```

সুডোকোড ৭.২

কমপ্লেক্সিটি

ক্রুসকাল অ্যালগরিদমের কমপ্লিক্সিটি $O(E \log E)$। গ্রাফে যদি এজের সংখ্যা তুলনামূলকভা বেশি হয় (Dense Graph) তাহলে প্রিম অ্যালগরিদম তুলনামূলকভাবে দ্রুত কাজ করে। এ কম থাকলে ক্রুসকাল অ্যালগরিদম দ্রুত কাজ করে।

প্রিম আর ক্রুসকাল অ্যালগরিদমের একটা গুরুত্বপূর্ণ পার্থক্য হলো গ্রাফে যদি একাধি কম্পোনেন্ট থাকে তাহলে ক্রুসকাল সবগুলো কম্পোননেন্টের জন্য মিনিমাম স্প্যানিং ট্রি বের ক দেবে, কিন্তু প্রিম প্রথম নোডটি যে কম্পোনেন্টে আছে শুধু সেটির মিনিমাম স্প্যানিং ট্রি বের ক দেবে।

২য় সেরা মিনিমাম স্প্যানিং ট্রি

অনেক সময় আমাদের ২য় সেরা মিনিমাম স্প্যানিং ট্রি বের করা দরকার হতে পারে। ২য় সে মিনিমাম স্প্যানিং ট্রি হলো এমন একটি স্প্যানিং ট্রি যার কস্ট মিনিমাম স্প্যানিং ট্রিয়ের থে বেশি কিন্তু অন্য যেকোনো স্প্যানিং ট্রি এর কস্টের সমান বা কম।

২য় সেরা মিনিমাম স্প্যানিং ট্রি বের করতে প্রথমে আগের মতো মিনিমাম স্প্যানিং ট্রি S_t বে করতে হবে। তারপর S_t থেকে প্রতিবার একটি করে এজ বাদ দিয়ে আবার সেই গ্রাফে মিনিমা স্প্যানিং ট্রি বের করতে হবে, এভাবে আরও E সংখ্যক মিনিমাম স্প্যানিং ট্রি পাওয়া যাবে, এদে মধ্যে যেটির মোট কস্ট সবচেয়ে কম সেটিই ২য় সেরা মিনিমাম স্প্যানিং ট্রি।

Scanned by CamScanner

অধ্যায় ৮ – টপোলজিকাল সার্টিং (Topological Sorting)

মনে কর, প্রতিদিন ঘুম থেকে উঠে তুমি কিছু কাজ কর যেগুলো হলো অফিসে যাওয়া, সকালে নাশতা করা, টিভিতে খেলা দেখা, কিছু ই-মেইলের উত্তর দেওয়া, বন্ধুদের সঙ্গে ডিনার করা ইত্যাদি। কাজগুলো কিন্তু তুমি যেকোনো ক্রমে করতে পারবে না, কিছু শর্ত মানতে হবে। যেমন অফিসে যাওয়ার আগে নাশতা করতে হবে, খেলা দেখার আগে অফিসে যেতে হবে, ডিনারে বসার আগে ই-মেইলের উত্তর দিতে হবে।

তুমি শর্তগুলোর তালিকা করে ফেললে,

1. সকালের নাশতা → অফিস (তারমানে নাশতা খেয়ে অফিসে যেতে হবে)
2. সুট-টাই পরা → অফিস (তার মানে সুট-টাই পরে অফিসে যেতে হবে)
3. অফিস → ই-মেইল
4. অফিস → ডিনার
5. অফিস → খেলা দেখা
6. ই-মেইল → ডিনার
7. ই-মেইল → খেলা দেখা
8. ডিনার → খেলা দেখা

তুমি এখন কোন কাজ কখন করবে? উল্টাপাল্টা ক্রম বা অর্ডারে করলে তোমার কাজ ভন্ডুল হয়ে যাবে, ই-মেইলের উত্তর না করে খেলা দেখতে বসলে তুমি ক্লায়েন্ট হারাবে, তাই অর্ডারিং খুব জরুরি। সর্টিং মানে হলো সাজানো। উপরের শর্তগুলোর মত কিছু শর্ত অনুসারে সাজানোকে বলা হয় টপোলজিকাল সর্টিং (Topological Sorting)।

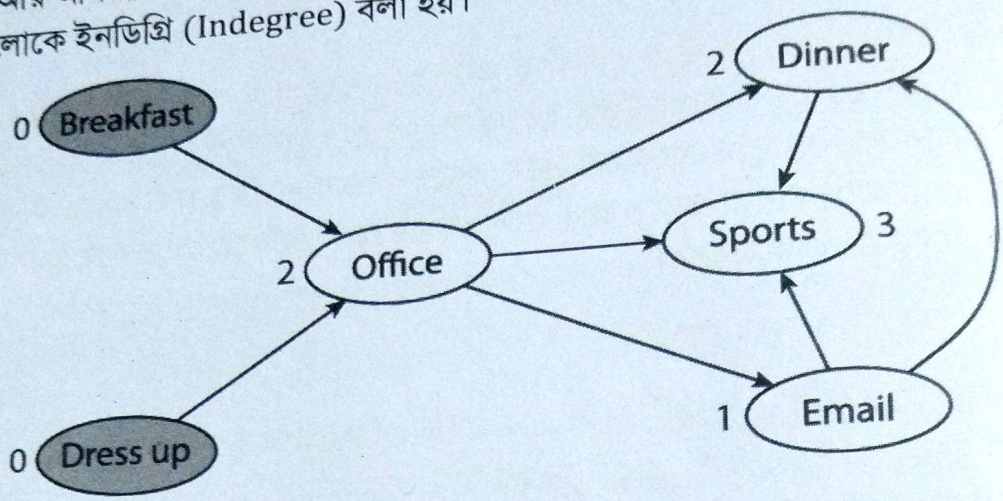
আমরা এই বইতে দুই ভাবে টপোলজিকাল সর্টিং শিখব। প্রথমে শিখব নোডগুলোর ইনডিগ্রি দেখে ব্রেডথ ফার্স্ট সার্চ ব্যবহার করে, পরবর্তীতে ডেপথ ফার্স্ট সার্চ অ্যালগরিদম শেখার পরে আরেকটি উপায়ে টপোলজিকাল সর্টিং করা শিখব।

আমরা প্রথমেই সমস্যাটিকে নিচের গ্রাফ দিয়ে মডেলিং করব। চিত্র ৮.১ এ প্রতিটি কাজ একটি করে নোড দিয়ে দেখানো হয়েছে। নাশতা বা ব্রেকফাস্ট থেকে অফিসের দিকে তীরচিহ্ন দিয়ে বোঝানো হচ্ছে যে, অফিসে আসার আগে ব্রেকফাস্ট করতে হবে।

আটটি শর্ত ছবিতে আটটি ডিরেক্টেড এজ দিয়ে দেখানো হয়েছে। প্রতিটি নোডের পাশে ছোট করে কিছু সংখ্যা দেখতে পাচ্ছো। যেমন অফিসের সঙ্গে 2, ইমেইলের সঙ্গে 1 ইত্যাদি। এগুলো দিয়ে বোঝানো হচ্ছে একটি কাজ অন্য কয়টি কাজের উপর সরাসরি নির্ভরশীল। যেমন ডিনারের আগে

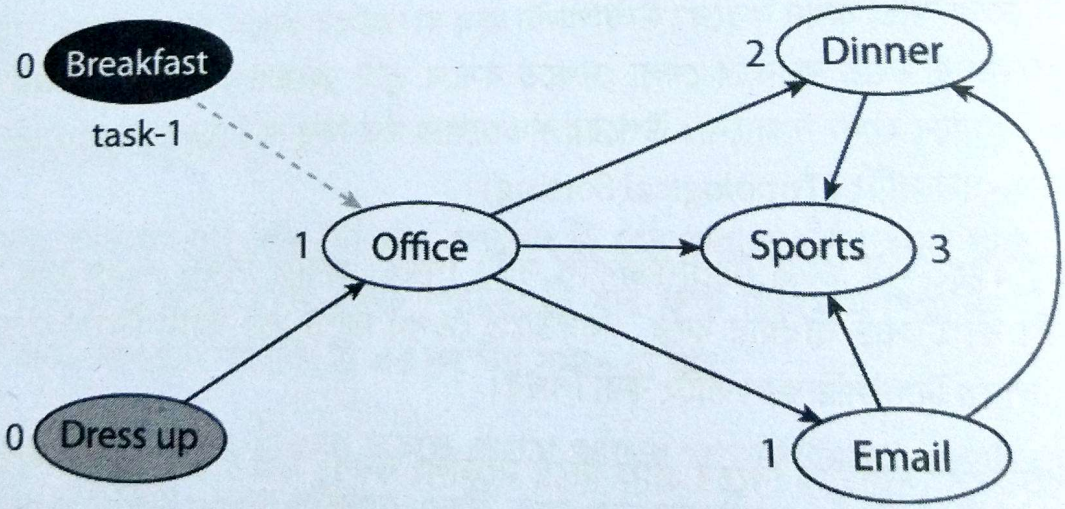
Scanned by CamScanner

তোমাকে অফিস ও ই-মেইল এই দুটি কাজ করতে হবে, ডিনার নোডটিতে দুটি তীরচিহ্ন প্রবেশ করেছে, আর আমরা পাশে লিখে দিয়েছি "2"। ঠিক এভাবে ই-মেইলের পাশে লেখা হয়েছে 1। এ সংখ্যাগুলোকে ইনডিগ্রি (Indegree) বলা হয়।



চিত্র ৮.১: কাজের ক্রমের শর্ত সম্বলিত একটি ওয়েটেড ডিরেক্টেড গ্রাফ

লক্ষ কর, ব্রেকফাস্ট এবং ড্রেসআপ কোনো কাজের উপর নির্ভরশীল নয়, তাই তাদের পাশে লেখা হয়েছে। তার মানে আমরা এ দুটি কাজের যেকোনোটি দিয়ে দিন শুরু করতে পারি। মনে কর তুমি নাশতা আগে খেতে চাও। নাশতা খেয়ে নেয়ার পর যেসব কাজ নাশতার উপর নির্ভরশীল ছিল তারা আর সেটার উপর নির্ভরশীল থাকল না, গ্রাফটি হয়ে গেল চিত্র ৮.২ এর মতো।

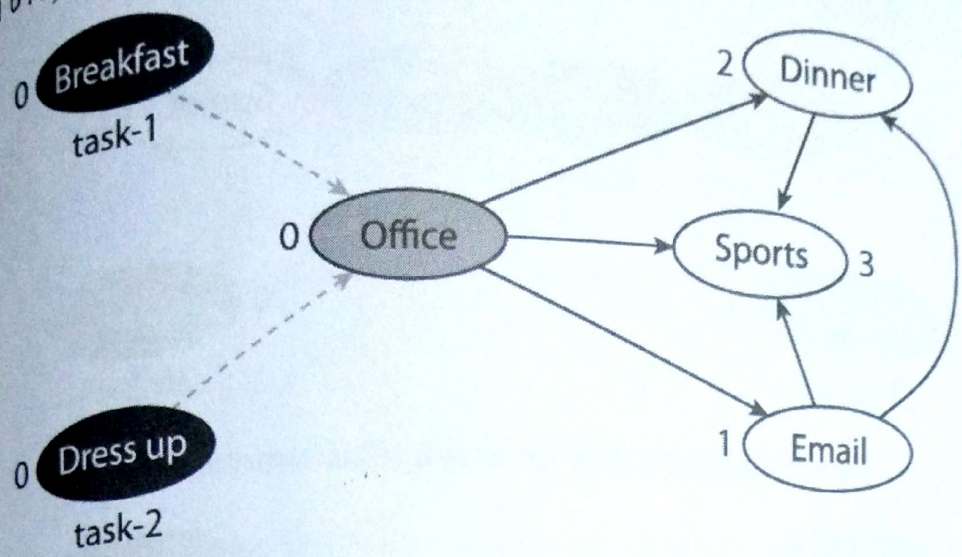


চিত্র ৮.২: নাস্তা সেরে নেওয়ার পরে তার উপরে অফিসে যাওয়ার নির্ভরশীলতা কমে গেল

ব্রেকফাস্ট থেকে অফিসের তীরচিহ্ন সরিয়ে দিয়েছি। এখন অফিস আর মাত্র একটি কাজের উপর নির্ভরশীল (আগে ছিল দুটির উপর)।

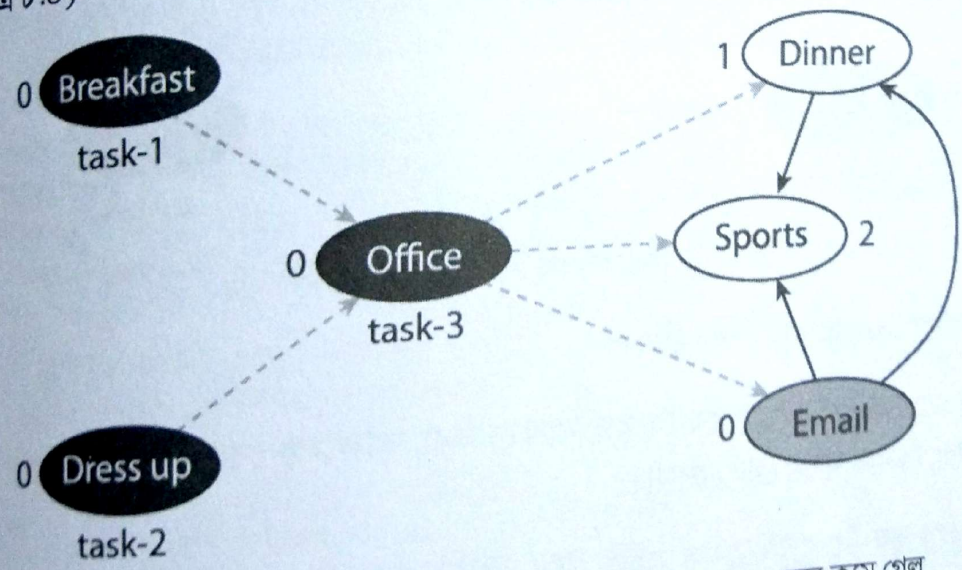
Scanned by CamScanner

এবার তোমাকে ড্রেসআপ করতে হবে, কারণ এখন একমাত্র এই কাজটিই কারো উপর নির্ভরশীল নয় (চিত্র ৮.৩)।



চিত্র ৮.৩: ড্রেস আপ করে ফেলার পর অফিসে যাওয়া আর কারও উপর নির্ভরশীল নয়

ড্রেসআপ থেকে অফিসের তীরচিহ্ন সরিয়ে দেওয়া হয়েছে। অফিসে যাওয়ার আগে আর কোনো কাজ নেই। এখন তুমি অফিসে যাওয়ার উপর যারা নির্ভরশীল তাদের তীরচিহ্নগুলো সরিয়ে দিতে পার (চিত্র ৮.৪)।

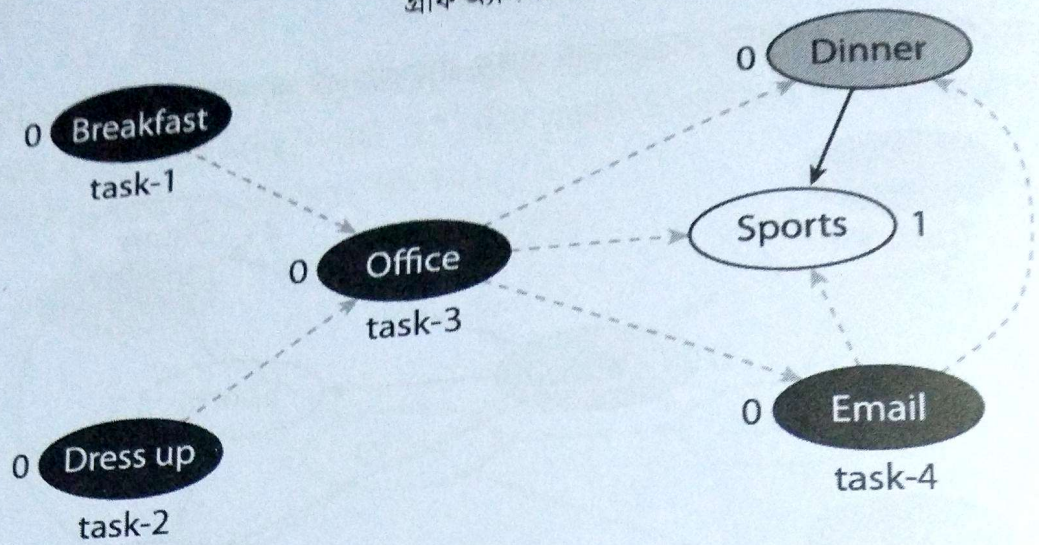


চিত্র ৮.৪: অফিসে যাওয়ার পরে বাকি কাজগুলোর নির্ভরশীলতা এক করে কমে গেল

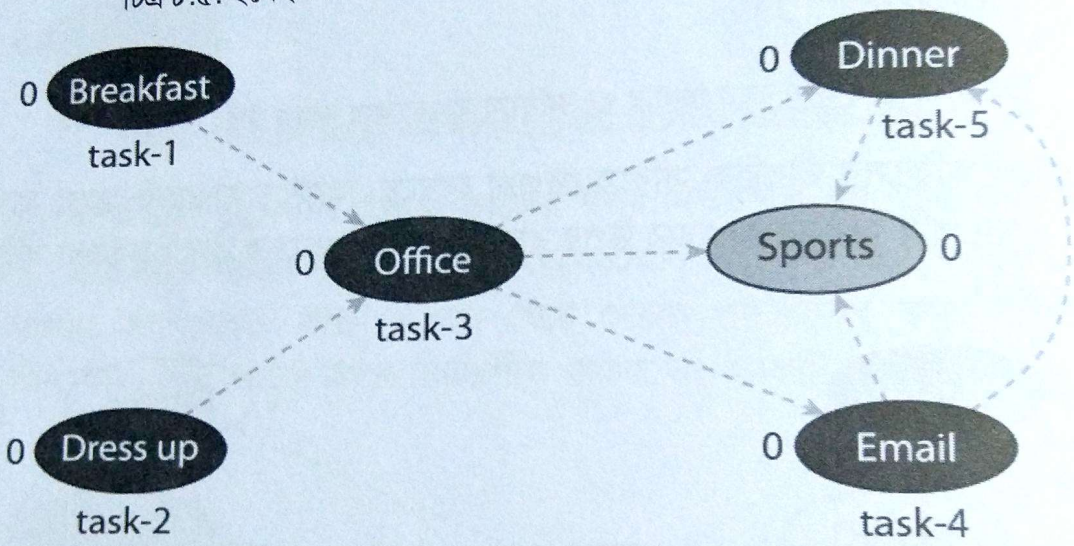
এখন "ই-মেইল" নোডের নির্ভরশীলতা 0 হয়ে গিয়েছে, গ্রাফটি আবার আপডেট করি (চিত্র ৮.৫)।

এবার ডিনার করার পালা (চিত্র ৮.৬)।

Scanned by CamScanner



চিত্র ৮.৫: ইমেইল চেক করার পরে ডিনার কাজটি এখন করা যেতে পারে



চিত্র ৮.৬: ডিনার করার পরে খেলা দেখতে বসা যায়

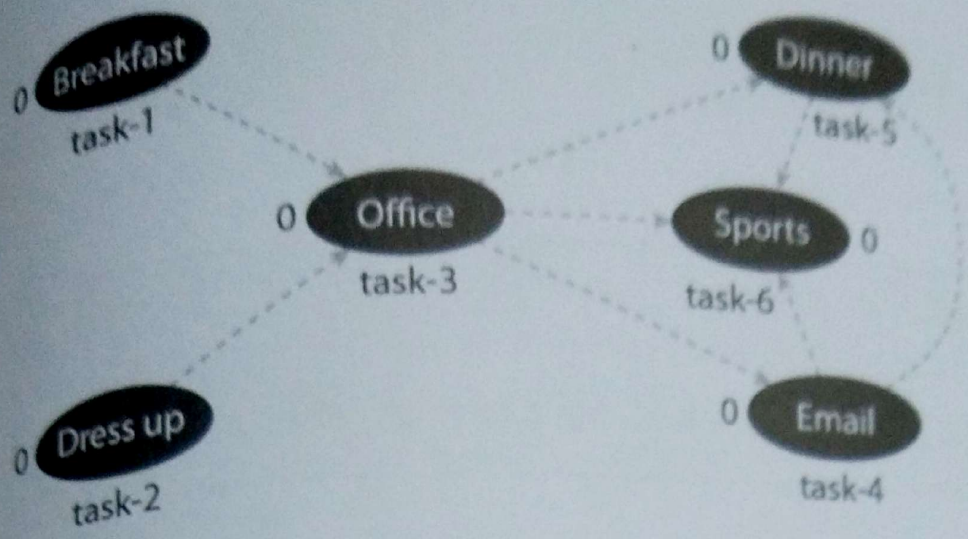
সবশেষে খেলা দেখতে বসা (চিত্র ৮.৭)।

এখন তুমি কাজের অর্ডারিং পেয়ে গিয়েছ, নাশতা করা, অফিসের পোষাক পড়া, অফিসে যাওয়া ইমেইল করা, ডিনার করা, খেলা দেখা।

এটিকেই বলা হয় টপোলজিক্যাল সর্টিং (Topological sorting) বা টপসর্ট। মূলত কাজের অর্ডারিং খুঁজে বের করতে এই অ্যালগরিদমটি ব্যবহার করা হয়। কম্পিউটার তার অভ্যন্তরে বিভিন্ন কাজের অর্ডারিং ঠিক করতে টপসর্ট ব্যবহার করে।

একটি ব্যাপার লক্ষ কর, গ্রাফে সাইকেল থাকলে সে গ্রাফের টপোলজিকাল সর্টিং করা সম্ভব নয়। একমাত্র ডিরেক্টেড অ্যাসাইক্লিক গ্রাফ বা DAG এর টপোলজিকাল সর্টিং করা সম্ভব।

Scanned by CamScanner



চিত্র ৮.৭: সবগুলো কাজের একটি নির্দিষ্ট ক্রম পাওয়া গেল

প্রোগ্রামিং টিপস

কোড লেখার সময় ছবির মতো বা সুডোকোডের মতো এজ মুছে ফেলার দরকার নেই, একটি ডিকশনারি ব্যবহার করে কোন নোডের ইনডিগ্রি কত সেটির হিসাব রাখবে এবং যে নোডের ইনকামিং এজ মুছতে হবে সেটার ইনডিগ্রি কমিয়ে দিবে।

```
procedure topsort(G):
  L ← Empty list that will contain the sorted elements
  S ← Set of all nodes with zero indegree

  while S is non-empty
    remove a node u from S
    L.add(u)

    for each adjacent node v of u do
      remove edge(u,v) from the graph
      if v has no other incoming edges
        S.add(v)
      end if
    end for

  end while

```


Scanned by CamScanner

```
return L
```

সুডোকোড ৮.১

কমপ্লেক্সিটি

প্রতিটি নোড এবং প্রতিটি এজ একবার করে কাজে লাগবে, কমপ্লেক্সিটি $O(V + E)$।

তুমি সেটের বদলে কিউ ব্যবহার করেও টপোলজিকাল সর্টিং করতে পারতে, তখন সুডোকোডটি অনেকটা বিএফএস অ্যালগরিদমের সুডোকোড এর মতো দেখাতো।

একটি গ্রাফের এক বা একাধিক টপলোজিকাল সর্টেড অর্ডার থাকতে পারে। সুডোকোডের ৫ নম্বর ধাপে যদি কখনও S সেটে একাধিক নোড থাকে তাহলে একাধিক অর্ডার আছে।

সবগুলো টপোলজিকাল অর্ডার বের করতে হলে ব্যাকট্র্যাকিং করতে হবে তবে সেটি নিয়ে এই বইতে আলোচনা করব না।

চিন্তা করার মত প্রশ্ন

সবগুলো টপলোজিকাল অর্ডার থেকে লেক্সিকোগ্রাফিকালি মিনিমাম অর্ডারিং (Lexicographically Minimum Ordering) টি কিভাবে বের করা যেতে পারে?

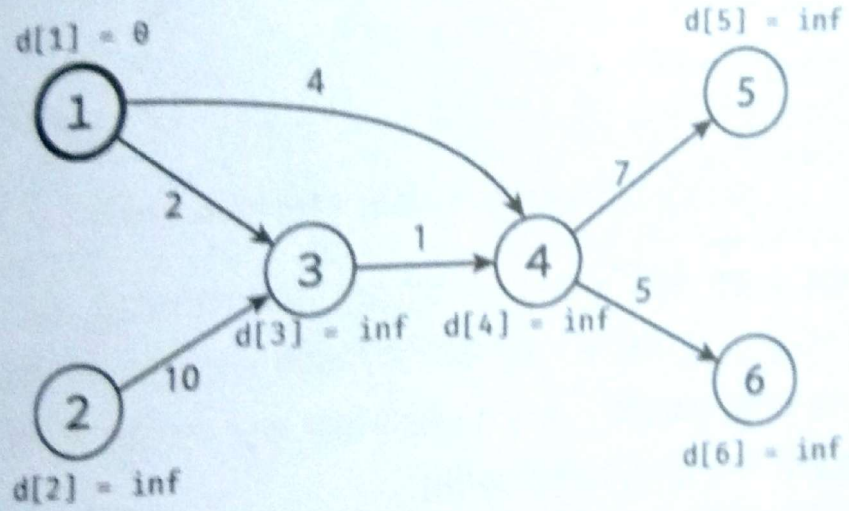
Scanned by CamScanner

অধ্যায় ৯ – বেলম্যান ফোর্ড অ্যালগরিদম (Bellman Ford Algorithm)

বেলম্যান ফোর্ড (Bellman Ford) অ্যালগরিদম গ্রাফে শর্টেস্ট পাথ বের করার একটি অ্যালগরিদম। এই অ্যালগরিদম একটি নোডকে সোর্স ধরে সেখান থেকে সব নোডের সংক্ষিপ্ততম বা শর্টেস্ট পাথ বের করতে পারে। আমরা একদম শুরুতে এই কাজ করার জন্য ব্রেডথ ফার্স্ট সার্চ শিখেছি। কিন্তু বিএফএস (BFS) যেহেতু ওয়েটেড গ্রাফে কাজ করে না তাই এরপর আমরা শিখেছি ডায়াক্সট্রা অ্যালগরিদম। এখন বেলম্যান ফোর্ড শিখব কারণ ডায়াক্সট্রা অ্যালগরিদম আবার নেগেটিভ ওয়েটের এজ আছে এমন গ্রাফে কাজ করে না।

আমরা ডায়াক্সট্রা অ্যালগরিদম শেখার সময় রিল্যাক্সিং নামে একটি ব্যাপার শিখেছিলাম। এখানেও সেটি কাজে লাগবে, তাই প্রথমে ডায়াক্সট্রার অধ্যায়টি শেষ করে আসতে হবে।

চিত্র ৯.১ এ আমরা নোড 1 থেকে শুরু করে প্রতিটি নোডে যাওয়ার শর্টেস্ট পাথ বের করতে চাই। আগের মতোই $d[\,]$ অ্যারেতে সোর্স নোড থেকে সবগুলো নোডের দূরত্ব রাখা থাকবে।



চিত্র ৯.১: একটি ওয়েটেড ডিরেক্টেড গ্রাফ

শুরুতে $d[1] = 0$ কারণ নোড 1 হলো সোর্স নোড। বাকি সবগুলোতে ইনফিনিটি রেখেছি কারণ আমরা এখনো জানি না শর্টেস্ট পাথের কস্ট কত।

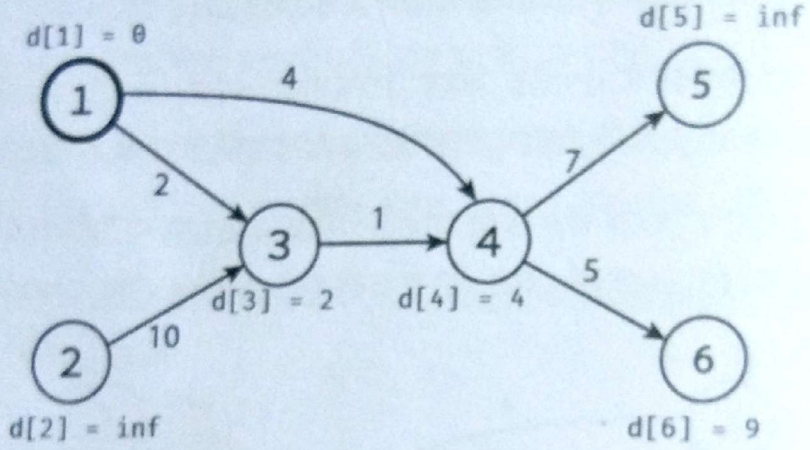
ডায়াক্সট্রা অধ্যায়ে তুমি এজ রিল্যাক্সিং কীভাবে করতে হয় এরই মধ্যে শিখে গেছ। এখন কাজ হলো সবগুলো এজকে যেকোনো অর্ডারে একবার করে রিল্যাক্সিং করা। একবার "গ্রাফ রিল্যাক্সিং" করার মানে হলো গ্রাফটির সবগুলো এজকে একবার করে রিল্যাক্সিং করা। আমি নিচের অর্ডারে রিল্যাক্সিং করতে চাই,

Scanned by CamScanner

ক্রম	1	2	3	4	5	6
এজ	[4→5]	[3→4]	[1→3]	[1→4]	[4→6]	[2→3]

তুমি চাইলে অন্য যেকোনো অর্ডারেও এজগুলো নিতে পারতে।

এখন চিন্তা কর এজগুলোকে একবার রিল্যাক্সিং করলে আমরা $d[\,]$ অ্যারেতে কী পাব? সোর্স নোড থেকে শুরু করে একটি এজ ব্যবহার করে অন্যান্য নোডে যাওয়ার শর্টেস্ট পাথের কস্ট (cost) আমরা পেয়ে যাব। চিত্র ৯.১ এ রিল্যাক্সিং করার পর $d[\,]$ এর মানগুলো আপডেট করে দাও। করার পর ছবিটা চিত্র ৯.২ এর মতো হবার কথা।



চিত্র ৯.২: একবার গ্রাফ রিল্যাক্সিং করার পরের অবস্থা

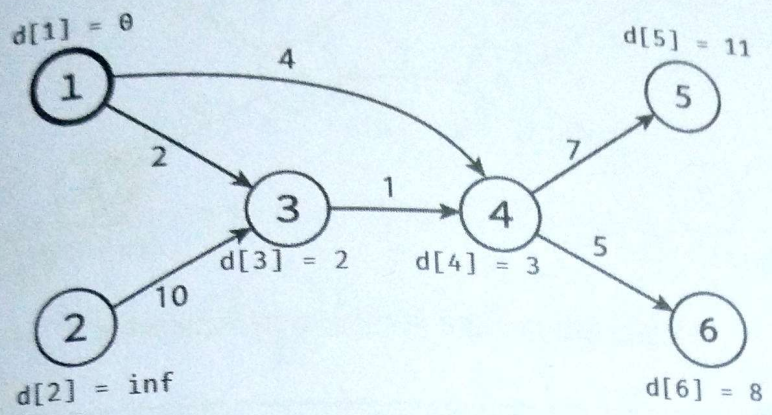
এজ রিল্যাক্সিং করার সময় কিছু নোডে d অ্যারের মান আপডেট করতে পারিনি কারণ $d[u] + cost[u][v] < d[v]$ শর্তটি পূরণ করেনি। বাকি এজগুলো আপডেট করার পর $d[\,]$ এর মান চিত্র ৯.২ এর মতো পেয়েছি। নোড 1 থেকে শুরু করে সর্বোচ্চ একটি এজ ব্যবহার করে সব নোডে যাবার শর্টেস্ট পাথ এখন আমরা জানি!

এখন দুটি এজ ব্যবহার করে সব নোডে যাবার শর্টেস্ট পাথের কস্ট বের করতে আরেকবার রিল্যাক্সিং করে ফেলি! আবারও যেকোনো অর্ডারে করা যাবে, তবে প্রথমে যে অর্ডারে করেছি সেভাবেই প্রতিবার করা কোড লেখার সময় সুবিধাজনক।

একটা ব্যাপার লক্ষ কর, নোড 1 থেকে নোড 6 এ যাওয়ার শর্টেস্ট পথে তিনটি এজ আছে ([1→3], [3→4], [4→6]) এবং পথের দৈর্ঘ্য $2 + 1 + 5 = 8$। মাত্র দুইবার রিল্যাক্সিং করলেও আমরা এখনই $distance[6]$ তে 8 পেয়ে গেছি, অথচ আমাদের এখন দুটি এজ ব্যবহার করে শর্টেস্ট পাথের কস্ট পাবার কথা। এটা নির্ভর করে তুমি কোন অর্ডারে এজ

Scanned by CamScanner

রিল্যাক্সিং করেছ তার উপর। সে কারণে নোড 5 এ যাওয়ার শর্টেস্ট পাথ 10 হলেও *distance*[5] এ এখনো 10 পাইনি।



চিত্র ৯.৩: দ্বিতীয়বার গ্রাফ রিল্যাক্সিং করার পরের অবস্থা

X সংখ্যক বার "গ্রাফ রিল্যাক্সিং" করলে X টি এজ ব্যবহার করে সোর্স নোড থেকে প্রতিটি নোডে যাওয়ার শর্টেস্ট পাথ তুমি নিশ্চিতভাবে পাবে। X এর থেকে বেশি এজের ব্যবহার করে প্রতিটি নোডে যাবার শর্টেস্ট পাথ তুমি X সংখ্যকবার গ্রাফ রিল্যাক্সিংয়ের পর পেতেও পার, নাও পেতে পার, সেটা এজের অর্ডারের ওপর নির্ভর করে।

X সংখ্যক বার "গ্রাফ রিল্যাক্সিং" করলে X টি এজ ব্যবহার করে সোর্স নোড থেকে প্রতিটি নোডে যাওয়ার শর্টেস্ট পাথ তুমি নিশ্চিতভাবে পাবে।

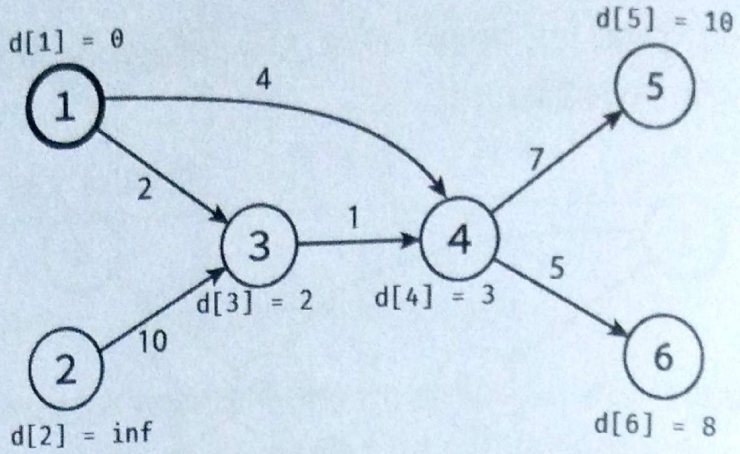
এখন ৩য় বারের মতো রিল্যাক্সিং করি (চিত্র ৯.৪)।

এবার শুধুমাত্র নোড 5 আপডেট হবে।

এরপরও আমরা আর যতই আপডেট করি, $distance[\,]$ অ্যারেতে কোনো পরিবর্তন হবে না, আমরা নোড 1 থেকে প্রতিটি নোডে যাওয়ার শর্টেস্ট পাথ পেয়ে গিয়েছি।

এখন স্বাভাবিকভাবেই প্রশ্ন আসবে যে রিল্যাক্সিং কয়বার করতে হবে? গ্রাফে যদি নোড nটি থাকে তাহলে এক নোড থেকে অন্য নোডে যেতে সর্বোচ্চ $(n-1)$টি এজ ব্যবহার করতে হবে। তার মানে কোনো নোড সর্বোচ্চ $(n-1)$ সংখ্যকবার আপডেট হতে পারে। তাই রিল্যাক্সিং করার লুপটিও চালাতে হবে $(n-1)$ সংখ্যকবার। তবে আমরা যেরকম উপরের গ্রাফে দেখেছি তিনবারের পরেই আর কোন নোড আপডেট করা যাচ্ছে না, সেরকম হলে আর নতুন করে রিল্যাক্সিং করার দরকার নেই।

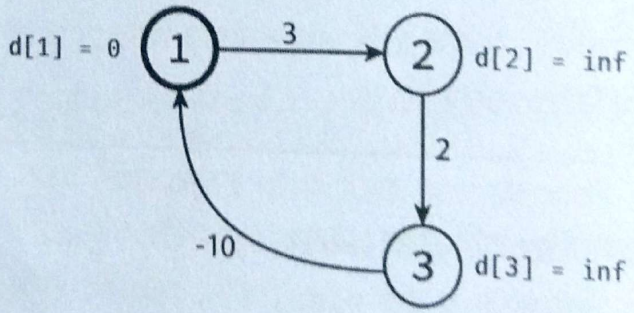
Scanned by CamScanner



চিত্র ৯.৪: তৃতীয়বার গ্রাফ রিল্যাক্সিং করার পরের অবস্থা

গ্রাফে যদি নোড nটি থাকে তাহলে এক নোড থেকে অন্য নোডে যেতে সর্বোচ্চ $(n-1)$টি এ ব্যবহার করতে হবে। তার মানে কোনো নোড সর্বোচ্চ $(n-1)$ সংখ্যকবার আপডেট হ পারে। তাই রিল্যাক্সিং করার লুপটিও চালাতে হবে সর্বোচ্চ $(n-1)$ সংখ্যকবার।

এখন নিচের মাত্র তিন নোডের গ্রাফটিতে (চিত্র ৯.৫) বেলম্যান ফোর্ড অ্যালগরিদম চালাও, অর্থ যতক্ষণ কোন নোড আপডেট করা যায় ততক্ষণ পুরো গ্রাফটা রিল্যাক্সিং কর।

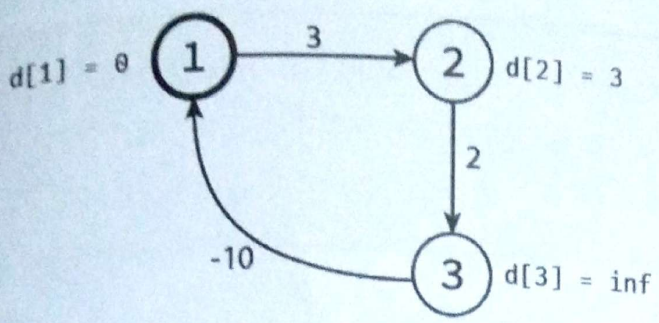


চিত্র ৯.৫: নেগেটিভ ওয়েট বিশিষ্ট ডিরেক্টেড গ্রাফ

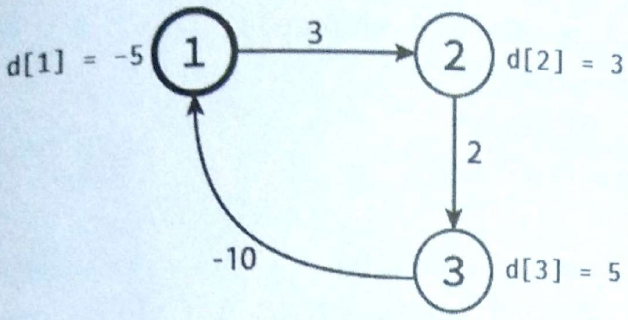
ক্রম	1	2	3
এজ	[2→3]	[1→3]	[3→1]

প্রথম বার রিল্যাক্সিং করার পর পাব চিত্র ৯.৬ এর মতো অবস্থা। দ্বিতীয় বার রিল্যাক্সিং করার প পাব চিত্র ৯.৭ এর মতো অবস্থা।

Scanned by CamScanner

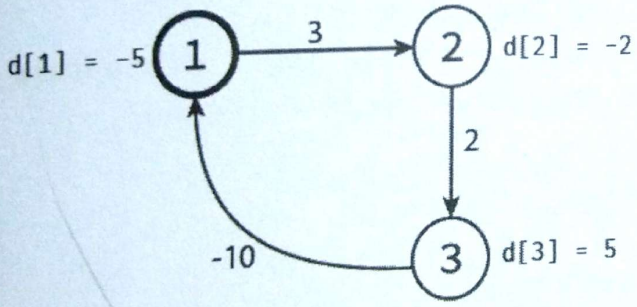


চিত্র ৯.৬: প্রথমবার গ্রাফ রিল্যাক্সিং করার পরের অবস্থা



চিত্র ৯.৭: দ্বিতীয়বার গ্রাফ রিল্যাক্সিং করার পরের অবস্থা

তিন নোডের গ্রাফে সোর্স নোড থেকে কোনো নোডে শর্টেস্ট পাথে যেতে দুটির বেশি এজ লাগবে না, তৃতীয় বার রিল্যাক্সিং করার চেষ্টা করলে কোনো নোড আপডেট হওয়ার কথা না। কিন্তু এই গ্রাফে আপডেট হচ্ছে (চিত্র ৯.৮)।



চিত্র ৯.৮: তৃতীয়বার গ্রাফ রিল্যাক্সিং করার পরের অবস্থা

এটা হচ্ছে কারণ [1→2→3→1] সাইকেলটির মোট ওয়েট নেগেটিভ $(3 + 2 - 10 = -5)$। তাই তুমি যতবার এই সাইকেলে ঘুরবে শর্টেস্ট পাথ তত ছোট হতে থাকবে। তাই নেগেটিভ সাইকেল থাকলে এবং সোর্স থেকে সেই নেগেটিভ সাইকেলে যাবার পথ থাকলে সোর্সের শর্টেস্ট পাথ আনডিফাইনড (Undefined) বা অসংজ্ঞায়িত। যদি $(n - 1)$ সংখ্যকবার গ্রাফ রিল্যাক্সিং করার পর দেখি যে nতম বারেও কোনো নোডের কস্ট আপডেট করা যায় তখন বুঝতে হবে আমরা নেগেটিভ সাইকেলে গিয়ে পরেছি, শর্টেস্ট পাথ বের করা সম্ভব নয়।

Scanned by CamScanner

যদি $(n-1)$ সংখ্যকবার গ্রাফ রিল্যাক্সিং করার পর দেখি যে nতম বারেও কোনো নোডের ক
আপডেট করা যায় তখন বুঝতে হবে আমরা নেগেটিভ সাইকেলে গিয়ে পরেছি, শর্টেস্ট পাথ
করা সম্ভব নয়।

আমাদের সুডোকোড তাহলে হবে এরকম,

```
procedure BellmanFord(G,source):
  distance[] ← infinity
  N ← number of nodes
  distance[source] ← 0

  for step from 1 to N-1 do
    for all edges from (u,v) in G.E do
      if distance[u] + cost[u][v] < distance[v]
        distance[v] ← distance[u] + cost[u][v]
      end if
    end for
  end for

  for all edges from (u,v) in G.E do
    if distance[u] + cost[u][v] < distance[v]
      return "Negative cycle detected"
    end if
  end for

  return distance
```

সুডোকোড ৯.১

কমপ্লেক্সিটি

বেলম্যান ফোর্ড অ্যালগরিদমের টাইম কমপ্লেক্সিটি $O(N*E)$। বিএফএস বা ডায়াক্সট্রা
যেভাবে পাথ প্রিন্ট করেছ ঠিক সেভাবে এখানেও পাথ প্রিন্ট করা যাবে।

Scanned by CamScanner

চিন্তা করার জন্য সমস্যা

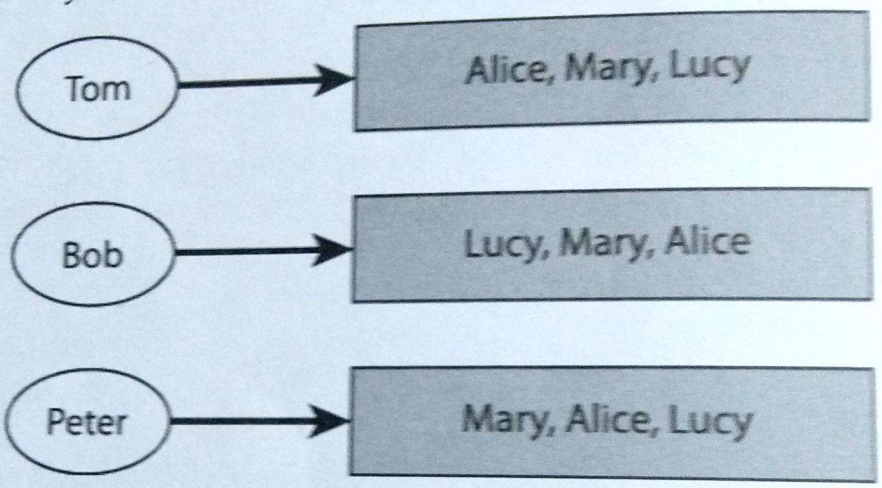
একটি গ্রাফে কোন কোন নোড নেগেটিভ সাইকেলের অংশ কীভাবে বের করবে? এই সমস্যাটি একটু কঠিন, অধ্যায় ১৩ তে স্ট্রংলি কানেক্টেড কম্পোনেন্ট অ্যালগরিদম শেখার পর এটি নিয়ে চিন্তা কর।

Scanned by CamScanner

অধ্যায় ১০ – স্টেবল ম্যারেজ (Stable Marriage)

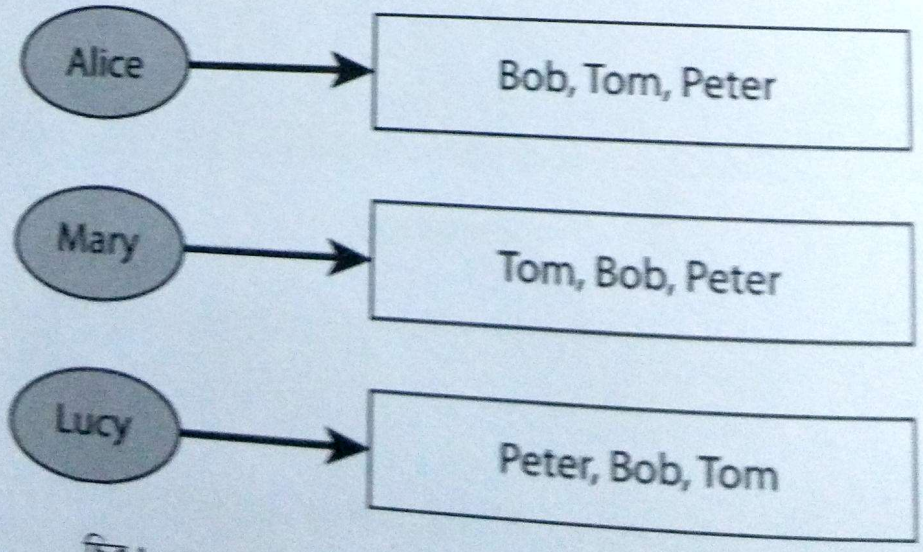
স্টেবল ম্যারেজ (Stable Marriage) খুবই মজার একটি সমস্যা। সমস্যাটি এরকম,

মনে করি n সংখ্যক ছেলে এবং n সংখ্যক মেয়ে আছে। এখন তাদের মধ্যে বিয়ে দিতে হবে এমন ভাবে যেন বিয়েগুলো "স্টেবল" হয়। প্রত্যেকের সঙ্গেই প্রত্যেকের বিয়ে দেওয়া সম্ভব তবে প্রতিটি ছেলে আর মেয়ের কিছু নিজস্ব পছন্দ আছে, প্রত্যেকেই চাইবে তার পছন্দের মানুষকে বিয়ে করতে। যদি ছেলে তিন জনের নাম $Tom, Bob, Peter$ আর মেয়ে তিন জনের নাম $Alice, Mary, Lucy$ হয় তাহলে ছেলেদের পছন্দের তালিকা হতে পারে চিত্র ১০.১ এর মতো।



চিত্র ১০.১: ছেলেদের পছন্দের ক্রমানুসারে মেয়েদের তালিকা

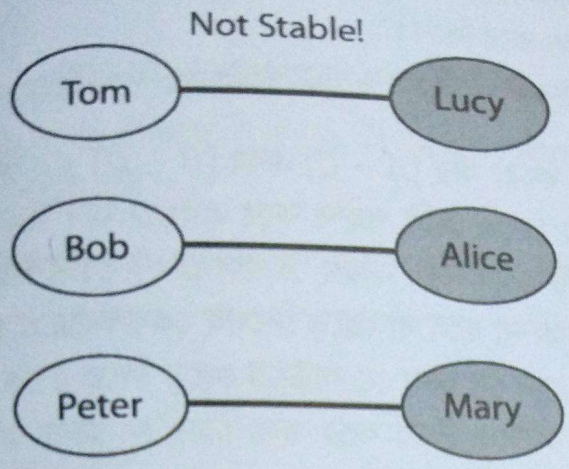
আবার মেয়েদের পছন্দের তালিকাটি হতে পারে চিত্র ১০.২ এর মতো।



চিত্র ১০.২: মেয়েদের পছন্দের ক্রমানুসারে ছেলেদের তালিকা

Scanned by CamScanner

এখন কী কম্বিনেশনে বিয়ে দিলে বিয়েগুলো স্টেবল হবে? আগে বোঝা দরকার স্টেবল বলতে কী বুঝাচ্ছি। ধরো চিত্র ১০.৩ মতো করে বিয়ে দেওয়া হলো।



চিত্র ১০.৩: বিয়ের একটি সম্ভাব্য কম্বিনেশন

এই ম্যাচিং/বিয়েটি স্টেবল না, কারণ টম, লুসির থেকে মেরিকে বেশি পছন্দ করে, আবার মেরি পিটারের থেকে টমকে বেশি পছন্দ করে। তাই টম আর মেরি বিয়ে ভেঙে একসঙ্গে চলে আসতে পারে। যদি A, B ছেলে আর C, D মেয়ে হয় আর $[A - C]$ এবং $[B - D]$ কে বিয়ে দেওয়া হয় তাহলে বিয়ে স্টেবল হবে না, যদি নিচের দুটি স্টেটমেন্টই সত্য হয়:

1. A যদি C এর থেকে D কে বেশি পছন্দ করে।
2. D যদি B এর থেকে A কে বেশি পছন্দ করে।

দুটি স্টেটমেন্টই সত্য হলে A আর D বিয়ে ভেঙে চলে আসবে! তবে যেকোনো একটি স্টেটমেন্ট মিথ্যা হলে বিয়ে স্টেবল হবে।

1962 সালে ডেভিড গেইল (David Gale) আর লয়েড শেইপলি (Lloyd Shapley) প্রমাণ করেন সমান সংখ্যক ছেলে আর মেয়ের জন্য সবসময় স্টেবল ম্যারেজ সমস্যার একটি সমাধান আছে। তারা খুব সহজ একটি অ্যালগরিদম আবিষ্কার করেন সমস্যাটি সমাধানের জন্য।

অ্যালগরিদমটি এরকম,

1. প্রথমে প্রতিটি অবিবাহিত ছেলে তার সবচেয়ে পছন্দের মেয়েটিকে প্রস্তাব পাঠাবে যাকে সে এখনো প্রস্তাব পাঠায়নি, মেয়েটি ইতিমধ্যে বিবাহিত হলেও সমস্যা নেই, একটি মেয়েকে একাধিক ছেলে প্রস্তাব পাঠাতে পারে। একটি ছেলে কখনও একটি মেয়েকে দুবার প্রস্তাব পাঠাবেনা।

Scanned by CamScanner

2. এবার প্রতিটি মেয়ে তাকে যারা প্রস্তাব পাঠিয়েছে তাদের মধ্যে থেকে যাকে সব থেকে পছন্দ তাকে বিয়ে করবে, বাকি সবাইকে বাতিল করে দিবে। মেয়েটি আগে অন্য কাউকে বিয়ে করে থাকলেও তাকে বাতিল করে দিবে।
3. এখনও কেউ অবিবাহিত থাকলে প্রথম ধাপের পুনরাবৃত্তি হবে।

অ্যালগরিদমটি কেন কাজ করে? ধরি $[A - C]$ এবং $[B - D]$ এর বিয়ে দেওয়া হয়েছে। তাহলে বিয়েটি ভেঙ্গে যাবে A যদি D কে বেশি পছন্দ করে এবং D যদি A কে বেশি পছন্দ করে। কিন্তু উপরের অ্যালগরিদমে সেটি সম্ভব নয়। কারণ, A যদি D কে বেশি পছন্দ করে তাহলে সে D কে আগে প্রস্তাব পাঠাবে, D রাজি না হলে বা ছেড়ে দিলেই কেবলমাত্র C কে প্রস্তাব পাঠাবে। D যদি A কে বেশি পছন্দ করে তাহলে সে অন্য যে কাউকে ছেড়ে দিয়ে A কে বিয়ে করবে। আর D যদি A কে বিয়ে না করে অন্য কাউকে বিয়ে করে তার মানে সে অন্য কাউকেই বেশি পছন্দ করে, এক্ষেত্রে বিয়ে ভাঙার সম্ভাবনা নেই।

এই অ্যালগরিমটি স্টেবল ম্যাচিং দিবে ঠিকই তবে অপটিমাল রেজাল্ট না-ও দিতে পারে। প্রতিটি ছেলের জন্য রেজাল্ট অপটিমাল হবে, কিন্তু মেয়েদের জন্য অপটিমাল নাও হতে পারে, অর্থাৎ এমন স্টেবল ম্যাচিং থাকতে পারে যেটাতে কোনো একটি মেয়ে আরও পছন্দের কাউকে বিয়ে করতে পারত। অর্থাৎ, যে প্রস্তাব পাঠাবে তার জন্য রেজাল্ট অপটিমাল হবে।

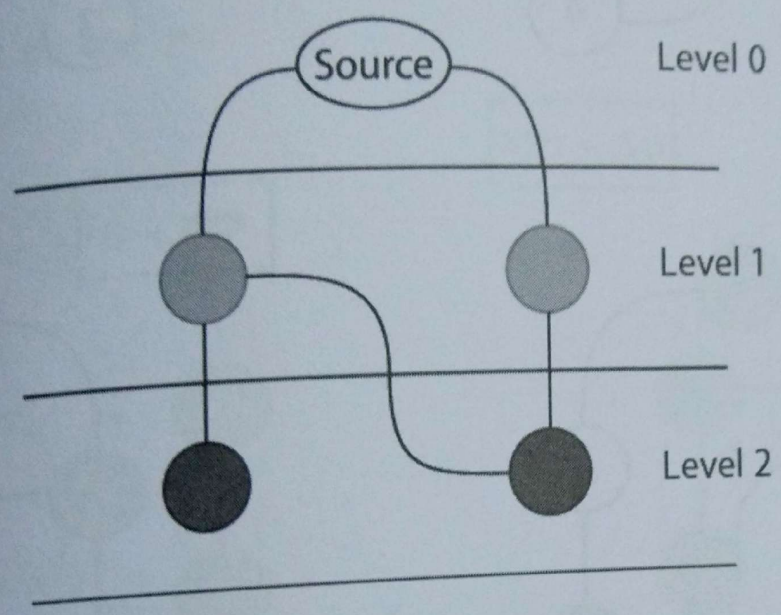
বাইপারটাইট ম্যাচিং অ্যালগরিদমকে কিছুটা পরিবর্তন করেও এই সমস্যাটি সমাধান করা যায়। ১৪ নম্বর অধ্যায়ে আমরা বাইপারটাইট ম্যাচিং নিয়ে জানব।

Scanned by CamScanner

অধ্যায় ১১ – ডেপথ ফার্স্ট সার্চ (Depth First Search)

এই অধ্যায়ে আমরা গ্রাফ সার্চিংয়ের আরেকটি পদ্ধতি শিখব যাকে বলা হয় ডেপথ ফার্স্ট সার্চ (Depth First Search) বা ডিএফএস (DFS)। ডিএফএস বুঝতে হলে রিকার্শন সম্পর্কে প্রাথমিক ধারণা অবশ্যই থাকতে হবে।

বিএফএসে আমরা গ্রাফটিকে লেভেল-বাই-লেভেল সার্চ করেছিলাম, চিত্র ১১.১ এর মতো করে।



চিত্র ১১.১: একটি গ্রাফে সোর্স থেকে বিভিন্ন লেভেলের নোড

এবার আমরা কোনো নোড পেলে সঙ্গে সঙ্গে সে নোড ধরে আরও গভীরে চলে যেতে থাকবো, যখন আর গভীরে যাওয়া যাবেনা তখন আবার আগের নোডে ফিরে এসে অন্য আরেক দিকে যেতে চেষ্টা করব। আগের মতই এক নোড কখনও একাধিকবার ভিজিট করব না।

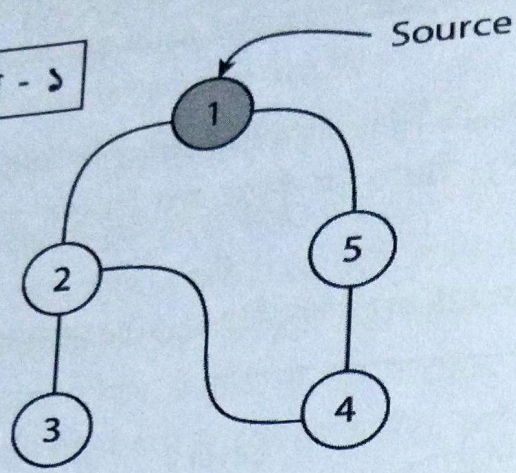
ডিএফএসে আমরা একটি নোডের তিন রকম অবস্থা তিনটি রঙ দিয়ে প্রকাশ করতে পারি:

- সাদা নোড = যে নোড এখনো খুঁজে পাইনি/ভিজিট করিনি।
- গ্রে বা ধুসর নোড = যে নোড ভিজিট করেছি কিন্তু নোডটি থেকে যেসব চাইল্ড নোডে যাওয়া যায় সেগুলো এখনো ভিজিট করে শেষ করিনি, অর্থাৎ নোডটিকে নিয়ে কাজ চলছে।
- কালো নোড = যে নোডের কাজ সম্পূর্ণ শেষ

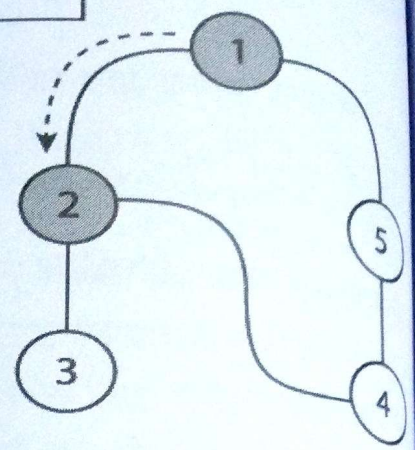
এবার আমরা একটা উদাহরণ দেখতে পারি (চিত্র ১১.২)।

Scanned by CamScanner

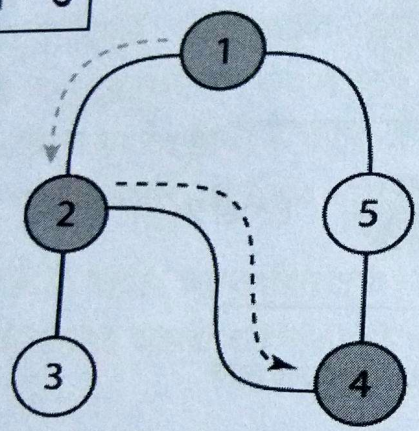
ধাপ - ১



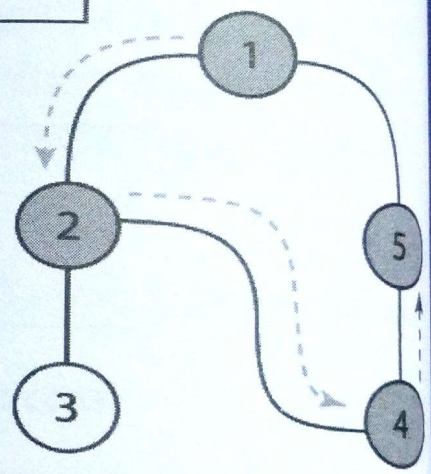
ধাপ - ২



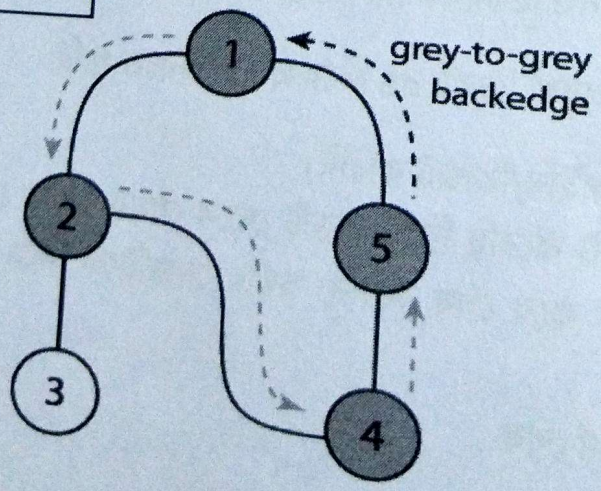
ধাপ - ৩



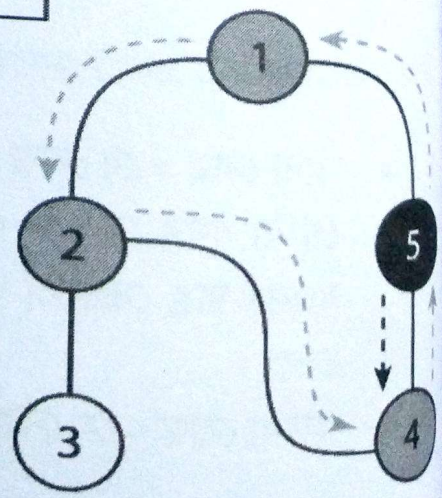
ধাপ - ৪



ধাপ - ৫

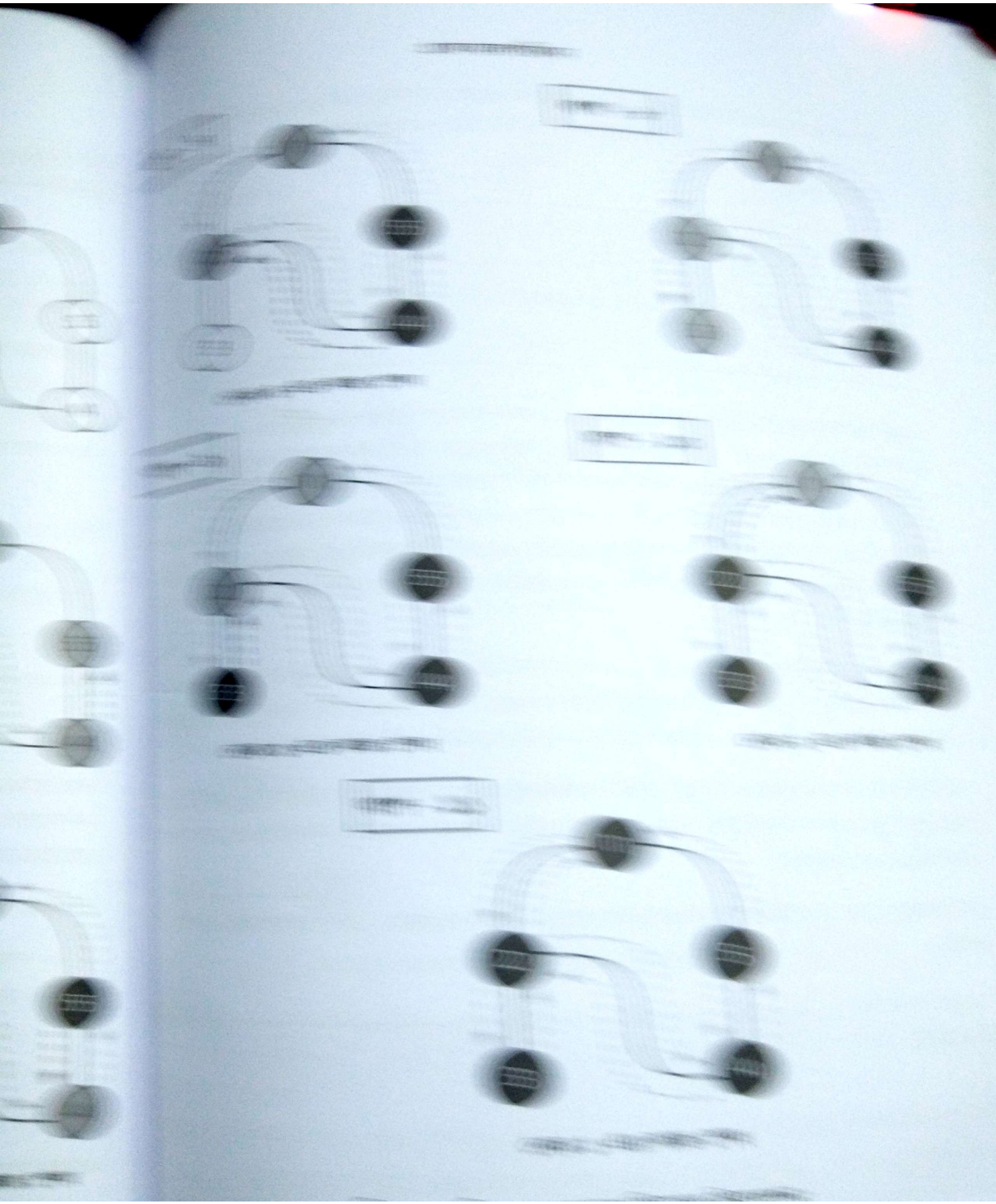


ধাপ - ৬



নোড ৫ এর কাজ শেষ

Scanned by CamScanner



Scanned by CamScanner

লক্ষ করো, আমরা শুধুমাত্র গ্রে নোড থেকে সাদা নোডে যাচ্ছি। এতে করে একটি ট্রি তৈরি হচ্ছে যাকে বলা হয় ডিএফএস ট্রি। গ্রে থেকে সাদা নোডের এজগুলোকে বলা হয় বলা হয় ট্রি এজ (Tree Edge)।

ডিএফএস ট্রি

শুধুমাত্র ট্রি এজ গুলো রেখে দিয়ে বাকি এজগুলো মুছে দিলে যে গ্রাফটি পাওয়া যায় তাকে বলা হয় ডিএফএস ট্রি।

আনডিরেক্টেড গ্রাফে ট্রি এজ ব্যতিত সকল এজকে বলা হয় ব্যাকএজ। $[5-1]$ কে বলা হয়েছে ব্যাকএজ (Back-edge), এর কারণ হলো তখনও নোড 1 এর কাজ চলছে এবং নোড 5 থেকে নোড 1 এ যাওয়া মানে এমন একটি নোডে ফিরে যাওয়া যাকে নিয়ে কাজ এখনও শেষ হয়নি, তার মানে অবশ্যই গ্রাফে একটি সাইকেল আছে। এ ধরনের এজকে ব্যাকএজ বলে। ডিএফএস অ্যালগরিদমে যদি কোনো সময় একটি গ্রে নোড u এবং আরেকটি গ্রে নোড v এর ভিতর এজ থাকে এবং ডিএফএস ট্রিতে u যদি v এর প্যারেন্ট নোড না হয়, তাহলে সে $[u-v]$ এজটি ব্যাকএজ।

ডিএফএস অ্যালগরিদমে সোর্স নোড এবং নোড ভিজিট করার অর্ডারের উপর নির্ভর করে সাইকেলে যেকোনো এজকে ব্যাকএজ হিসাবে পাওয়া যেতে পারে, যেমন নোড 1 থেকে আগে নোড 2 তে না গিয়ে নোড 5 এ গেলে পরে $[2-1]$ কে ব্যাকএজ হিসাবে পাওয়া যেত।

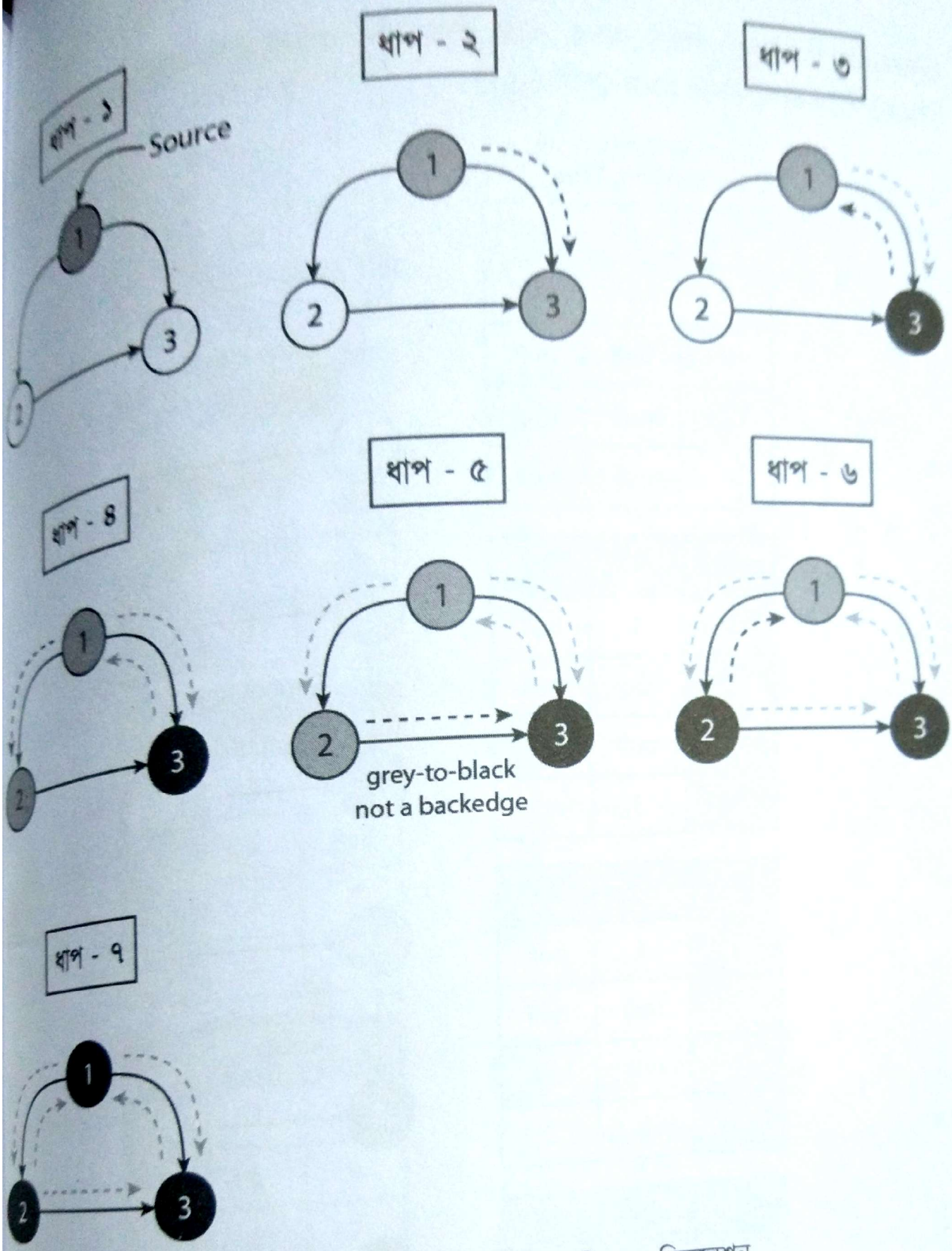
আনডিরেক্টেড গ্রাফের ক্ষেত্রে আগে ভিজিট করা হয়েছে এমন কোনো নোডে ফিরে গেলেই সেটি ব্যাকএজ, রঙ চেক না করলেও হয়। তবে ডিরেক্টেড গ্রাফের ক্ষেত্রে অবশ্যই রঙ চেক করতে হবে। চিত্র ১১.৩ দেখো।

এই গ্রাফে $[2 \rightarrow 3]$ এজটিকে ব্যাকএজ বলা যাচ্ছে না, কারণ নোড 3 এর কাজ আগেই শেষ হয়ে গেছে।

কমপ্লেক্সিটি:

বিএফএসের মতই প্রতিটি নোডে একবার করে গিয়েছি, প্রতিটি এজে একবার গিয়েছি। তাহলে কমপ্লেক্সিটি হবে $O(V+E)$ যেখানে V হলো নোড সংখ্যা এবং E হলো এজ সংখ্যা।

Scanned by CamScanner



চিত্র ১১.৩: ডিরেক্টেড গ্রাফে ডিএফএস অ্যালগরিদমের সিমুলেশন

এবার আমরা ডিসকভারি টাইম এবং ফিনিশিং টাইম এর ধারণার সঙ্গে পরিচিত হব। এই ধারণাগুলো আর্টিকুলেশন পয়েন্ট, স্ট্রংলি কানেক্টেড কম্পোনেন্ট ইত্যাদি অ্যালগরিদম শেখার সময় কাজে লাগবে।

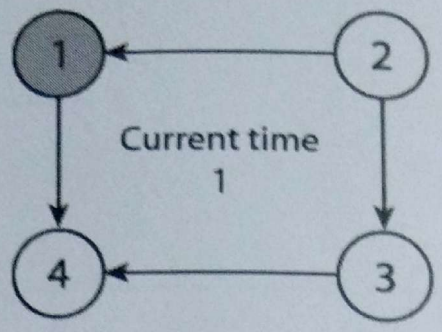
এখন আমরা ডিএফএস চালানোর সময় একটি স্টপওয়াচ চালু করে দিব আর কোনো নোড নিয়ে কাজ শুরু করলে ঘড়ি দেখে নোডটির স্টার্টিং টাইম (Starting Time) / ডিসকভারি টাইম

Scanned by CamScanner

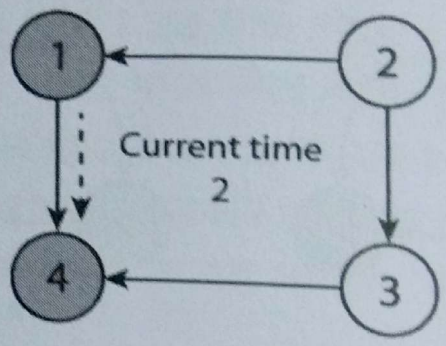
(Discovery Time) লিখে রাখব, কাজ শেষ হলো নোডটির ফিনিশিং টাইম (Finishing Time) লিখে রাখব। চিত্র ১১.৪ দেখো।

d[ ] = Discovery Time
f[ ] = Finishing Time

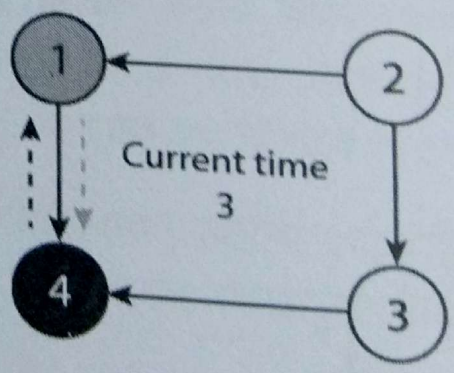
Node	d[ ]	f[ ]
1	1	null
2	null	null
3	null	null
4	null	null



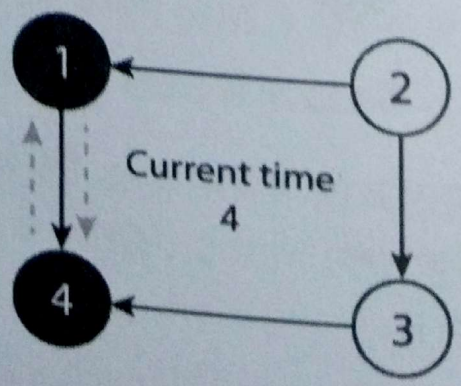
Node	d[ ]	f[ ]
1	1	null
2	null	null
3	null	null
4	2	null



Node	d[ ]	f[ ]
1	1	null
2	null	null
3	null	null
4	2	3

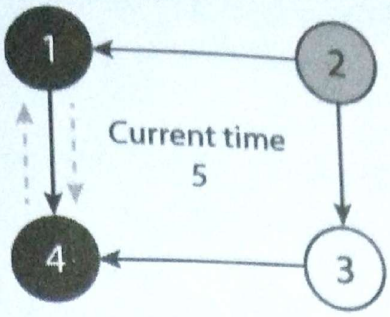


Node	d[ ]	f[ ]
1	1	4
2	null	null
3	null	null
4	2	3

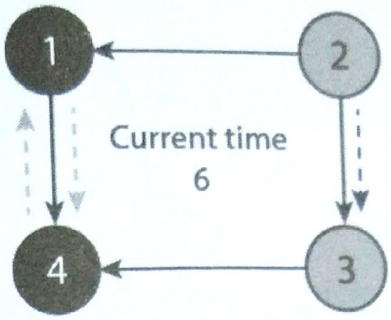


Scanned by CamScanner

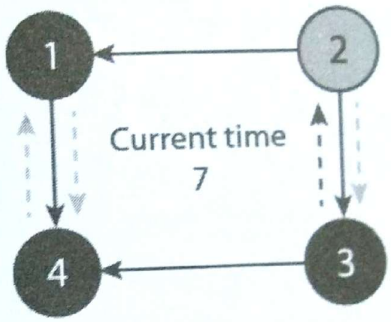
Node	d[]	f[]
1	1	4
2	5	null
3	null	null
4	2	3



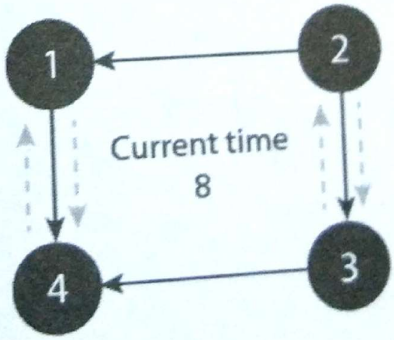
Node	d[]	f[]
1	1	4
2	5	null
3	6	null
4	2	3



Node	d[]	f[]
1	1	4
2	5	null
3	6	7
4	2	3



Node	d[]	f[]
1	1	4
2	5	8
3	6	7
4	2	3



ফিনিশিং টাইম অনুযায়ী সর্ট করে পাই: নোড 2, নোড 3, নোড 1, নোড 4

চিত্র ১১.৪: ডিএফএস অ্যালগরিদম ব্যবহার করে ডিসকভারি টাইম ও ফিনিশিং টাইম নির্ণয়

গ্রাফটিতে যদি কোনো সাইকেল না থাকে ফিনিশিং টাইম দেখে আমরা সহজেই টপোলজিকাল সর্ট করতে পারি। যে নোডটি সবার আগে আসবে তার ফিনিশিং টাইম অবশ্যই সবচেয়ে বেশি

Scanned by CamScanner

হবে, কারণ প্রথম নোডের ওপর নির্ভরশীল সব নোড ঘুরে আসার পরে সে নোডের ফিনিশিং টাইম গণনা করা হচ্ছে।

প্রোগ্রামিং টিপস

ডেপথ ফার্স্ট সার্চ সাধারণত রিকার্শন ব্যবহার করে ইম্প্লিমেন্টেশন করা হয়। যদি ডিএফএস ট্রি এর গভীরতা অনেক বেশি হয় তাহলে এটি স্ট্যাক ওভারফ্লো (Stack Overflow) এরর দিবে। ডিএফএস অ্যালগরিদম ব্যবহারের সময় এই ব্যাপারটি অবশ্যই মাথায় রাখতে হবে।

চলো সুডকোডটি দেখি,

```
procedure DFS(G, source):
  U ← source
  time  ← time+1
  d[u]  ← time
  color[u]  ← GREY

  for all edges from u to v in G.adjacentEdges(v) do
    if color[v] = WHITE
     DFS(G,v)
    end if
  end for

  color[u] ← BLACK
  time ← time+1
  f[u] ← time

  return
```

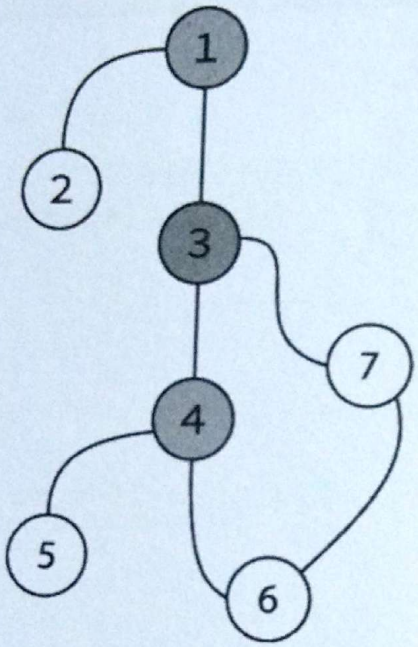
সুডোকোড ১১.১

Scanned by CamScanner

অধ্যায় ১২ – আর্টিকুলেশন পয়েন্ট (Articulation Point) এবং ব্রিজ (Bridge)

১২.১ – আর্টিকুলেশন পয়েন্ট (Articulation Point)

আর্টিকুলেশন পয়েন্ট হলো আনডিরেক্টেড গ্রাফের এমন একটি নোড যেটি গ্রাফ থেকে মুছে ফেললে বাকি গ্রাফটুকু একাধিক কম্পোনেন্টে ভাগ হয়ে যায় (চিত্র ১২.১)।



চিত্র ১২.১: একটি আনডিরেক্টেড গ্রাফে আর্টিকুলেশন পয়েন্টগুলো দেখানো হয়েছে

চিত্র ১২.১ এ নোড 1, নোড 3 অথবা নোড 4 এবং সেই নোডের অ্যাডজেসেন্ট এজগুলোকে মুছে দিলে গ্রাফটি একাধিক ভাগ হয়ে যাবে, তাই নোড 1, নোড 3 ও নোড 4 হলো এই গ্রাফের আর্টিকুলেশন পয়েন্ট। আর্টিকুলেশন পয়েন্টকে অনেকে **কাট-নোড (Cut Node)** বা **ক্রিটিকাল পয়েন্ট (Critical Point)** বা **আর্টিকুলেশন নোডও** বলে।

আর্টিকুলেশন পয়েন্ট বের করার একটি খুব সহজ উপায় হলো, একটি করে নোড গ্রাফ থেকে মুছে দিয়ে দেখো যে গ্রাফটি একাধিক কম্পোনেন্টে বিভক্ত হয়ে গিয়েছে কি না। কম্পোনেন্ট সংখ্যা ডিএফএস বা বিএফএস অ্যালগরিদম দিয়ে খুব সহজে বের করা যায়। এই পদ্ধতিতে V সংখ্যকবার ডিএফএস/বিএফএস অ্যালগরিদম চালাতে হবে যেখানে V হলো নোড সংখ্যা, মোট কমপ্লেক্সিটি $O(V * (V + E))$ বা $O(V^3)$ কারণ সর্বোচ্চ এজ সংখ্যা V^2। নিচের সুডোকোডটি দেখ,

Scanned by CamScanner

```
procedure articulationPointNaive(G):
  Articulation_points ← []

  for all nodes u in G do
    G.removeNode(u)

    if get_number_of_component(G) > 1
      Articulation_points.add(u)
    end if

    G.addNode(u)
  end for

  return Articulation_points
```

সুডোকোড ১২.১

আর্টিকুলেশন পয়েন্ট বের করার একটি খুব সহজ উপায় হলো, একটি করে নোড গ্রাফ থেকে মুছে দিয়ে দেখো যে গ্রাফটি একাধিক কম্পোনেন্টে বিভক্ত হয়ে গিয়েছে কি না।

এখন আমরা একবারমাত্র ডিএফএস অ্যালগরিদম চালিয়ে আর্টিকুলেশন পয়েন্ট বের করব। এই অ্যালগরিদম শেখার জন্য ডিএফএস অ্যালগরিদমের ডিসকভারি/ফিনিশিং টাইম এবং ট্রি এজ ও ব্যাকএজ নিয়ে ধারণা থাকতে হবে।

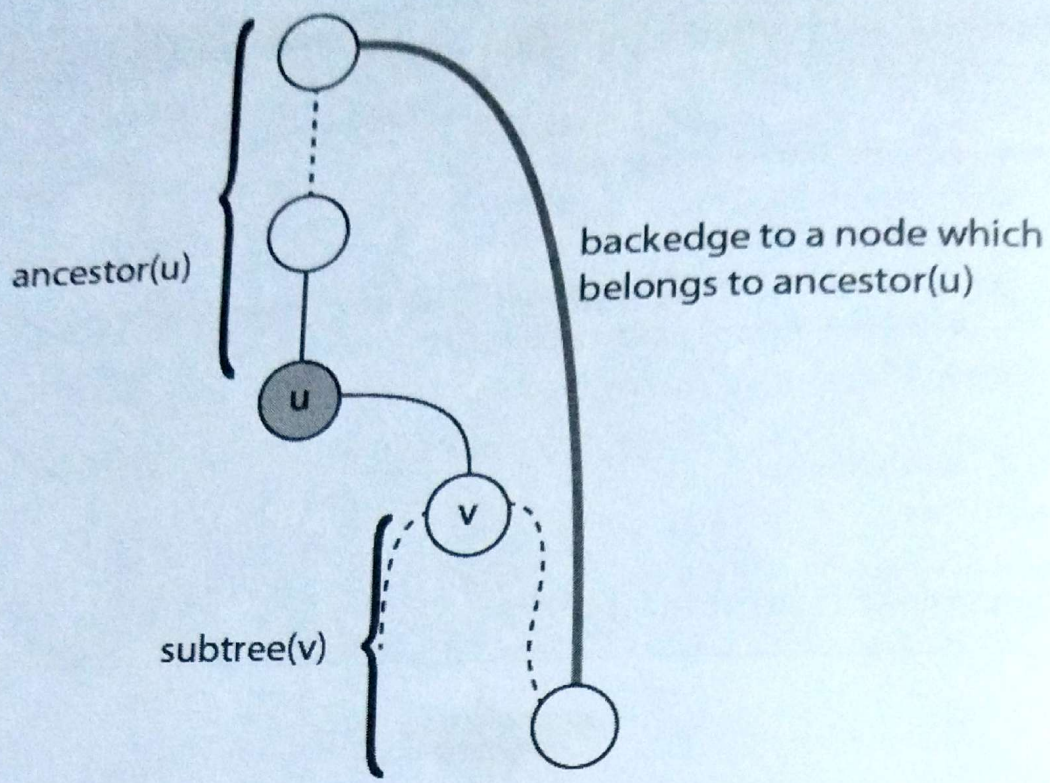
একটি গ্রাফে ডিএফএস অ্যালগরিদম চালালে যেসব এজ পাওয়া যায় সেগুলো নিয়ে তৈরি হয়ে ডিএফএস ট্রি (শুধুমাত্র ট্রি এজগুলো নিব, ব্যাকএজ নিব না)। দুটি কেস থাকতে পারে। যদি একটি নোড ট্রিয়ের রুট হয় তাহলে একভাবে কাজ করব, রুট না হলে আরেকভাবে কাজ করব।

একটি নোড u যদি ট্রিয়ের রুট হয় এবং ডিএফএস ট্রিতে নোডটির একাধিক চাইল্ড নোড থাকে তাহলে নোডটি আর্টিকুলেশন পয়েন্ট। এটি হলো প্রথম কেস।

রুট ছাড়া বাকি নোডের জন্য কাজটি একটু জটিল।

ডিএফএস ট্রিয়ের একটি এজ $[u - v]$ এর কথা চিন্তা কর। রুট থেকে নোড u তে আসার পথে যেসব নোড তুমি ভিজিট করেছ তাদেরকে বলবো $ancestor(u)$। এখন নোড v যে সাবট্রিয়ের রুট সেই সাবট্রিয়ের সবগুলো নোডের সেটকে আমরা বলব $subtree(v)$।

Scanned by CamScanner



চিত্র ১২.২: ডিএফএস ট্রিতে $[u - v]$ এজের জন্য অ্যানসেস্টর ও সাবট্রি

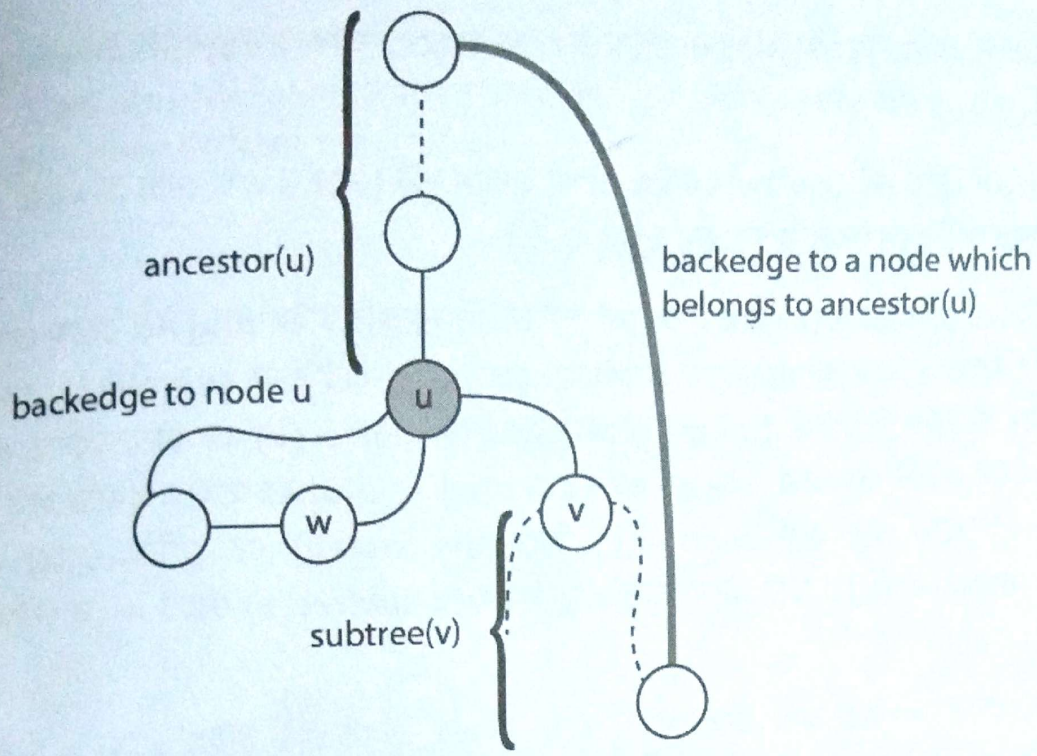
এখন নোড u একটি আর্টিকুলেশন পয়েন্ট হবে যদি মূল গ্রাফে নোড u কে মুছে দিলে $subtree(v)$ এর নোডগুলো একটি আলাদা কম্পোনেন্টে পরিণত হয়। $subtree(v)$ আলাদা কম্পোনেন্টে পরিণত হবে যদি না মূল গ্রাফে $subtree(v)$ এর কোনো নোড থেকে $ancestor(u)$ তে একটি ব্যাকএজ থাকে। যদি ব্যাকএজ থাকে তাহলে নোড u এবং তার অ্যাডজেসেন্ট এজগুলো মুছে গেলেও $ancestor(u)$ থেকে ব্যাকএজ দিয়ে $subtree(v)$ তে পৌঁছানো যাচ্ছে, নতুন কম্পোনেন্ট তৈরি হচ্ছে না।

নোড u এর যেকোনো একটি চাইল্ড নোড v এর জন্য যদি $subtree(v)$ থেকে $ancestor(u)$ তে পৌঁছানো না যায় , তাহলে নোড u আর্টিকুলেশন পয়েন্ট, নোড u কে মুছে দিলে সেইসব $subtree(v)$ নতুন কম্পোনেন্টে পরিণত হবে যাদের সঙ্গে $ancestor(u)$ এর কোনো ব্যাকএজ সংযোগ নেই।

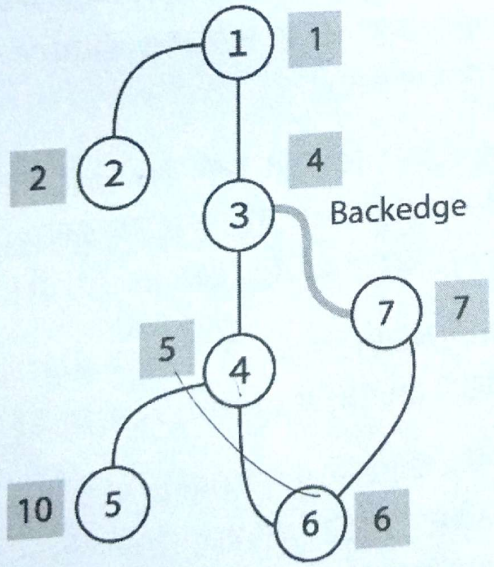
চিত্র ১২.৩ এ $subtree(v)$ যদিও ব্যাকএজ দিয়ে $ancestor(u)$ এর সঙ্গে সংযুক্ত, $subtree(w)$ থেকে $ancestor(u)$ তে ব্যাকএজ নেই। তাই নোড u একটি আর্টিকুলেশন পয়েন্ট।

এবার প্রথম গ্রাফটিতে (চিত্র ১২.১) ফিরে আসি। গ্রাফের নোডগুলো 1, 2, 3, 4, 6, 7, 5 এই অর্ডারে ভিজিট করলে আমরা প্রতিটি নোডের যে ডিসকভারি টাইম পাব সেটি পাশে ছোট করে লেখা হয়েছে (চিত্র ১২.৪)।

Scanned by CamScanner



চিত্র ১২.৩: $subtree(w)$ থেকে $ancestor(u)$ তে কোনো ব্যাকএজ নেই



চিত্র ১২.৪: চিত্র ১২.১ গ্রাফে নির্দিষ্ট ক্রমে ভিজিট করার পর পাওয়া ডিসকভারি টাইম

ডিসকভারি টাইম কীভাবে বের করতে হয় না বুঝলে ডেপথ ফার্স্ট সার্চ অধ্যায় দেখো। $d[\,]$ দিয়ে আমরা ডিসকভারি টাইম বোঝাবো।

Scanned by CamScanner

গ্রাফের ব্যাকএজ [3–7] আলাদা ভাবে দেখানো হয়েছে। বাকি এজগুলো ডিএফএস ট্রিয়ের অংশ। 1 হলো রুট নোড।

ডিএফএস ট্রিতে রুট নোড 1 এর চাইল্ড সংখ্যা এখানে দুটি (নোড 2 এবং নোড 3)। তাই নোড 1 একটি আর্টিকুলেশন পয়েন্ট (কেস 1)।

লক্ষ করো, $ancestor(u)$ এর যেকোনো নোডের ডিসকভারি টাইম $d[u]$ এর থেকে ছোটো। আবার নোড u এর অ্যাডজেসেন্ট যেকোনো এজ $[u-v]$ এর জন্য $subtree(v)$ এর সব নোডের ডিসকভারি টাইম $d[u]$ এর থেকে বড়। এখন $subtree(v)$ এর কোনো নোড থেকে যদি এমন একটি ব্যাকএজ $[v-w]$ থাকে যেন $d[w] < d[u]$ হয় তাহলে বুঝতে হবে তুমি $[u-v]$ এজ পার হয়ে $subtree(v)$ দিয়ে $ancestor(u)$ তে পৌঁছে গেছো এবং $w \in ancestor(u)$ । তার মানে নোড u মুছে দিলেও $subtree(v)$ থেকে w তে পৌঁছানো যাবে।

যেমন নোড 4 এর কথা চিন্তা কর। নোড 4 এর ডিসকভারি টাইম $d[4] = 5$ এবং $ancestor(4) = \{1, 3\}$। এখন $[4-6]$ এজটির কথা ভাবি। $subtree(6)$ এ একটি ব্যাকএজ $[7-3]$ আছে এবং $d[3] = 4$ যা $d[4]$ এর থেকে ছোট। তারমানে $3 \in ancestor(4)$। তাহলে তুমি নোড 4 মুছে দিলেও $subtree(6)$ ব্যাকএজের মাধ্যমে $ancestor(4)$ এর সঙ্গে সংযুক্ত থাকবে।

এবার আমরা আরেকটি ভ্যারিয়েবল ডিফাইন করব $low[u]$। মনে কর $subtree(u)$ এবং $subtree(u)$ এর সঙ্গে ব্যাকএজ দিয়ে সংযুক্ত সবগুলো নোডের একটি সেট বানানো হল, সেটটি হলো $\{x1, x2, \ldots, xm\}$। তাহলে $low[u]$ হবে $min(d[x1], d[x2], \ldots, d[xm])$।

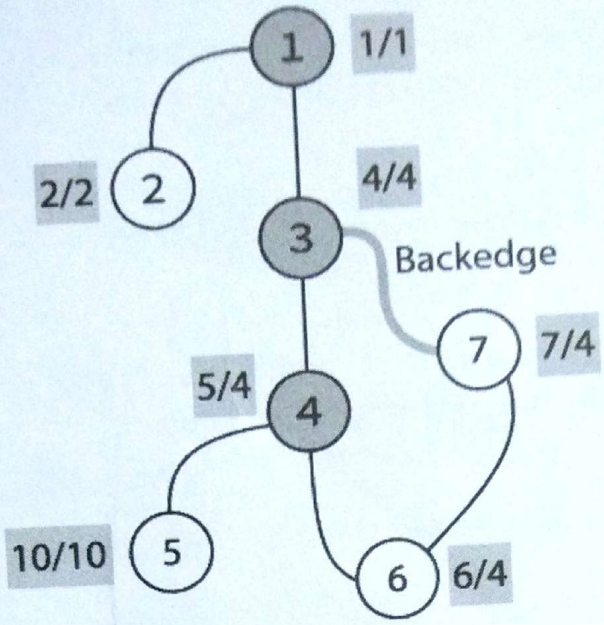
যেমন নোড 4 এর জন্য $subtree(u) = \{5, 6, 7\}$ এবং $subtree(u)$ এর সঙ্গে ব্যাকএজ দিয়ে যুক্ত আছে নোড 3। তাহলে $low[u] = min(d[5], d[6], d[7], d[3]) = 4$।

এখন চিন্তা কর, কোনো একটি এজ $[u-v]$ এর জন্য $d[u] > low[v]$ হওয়ার অর্থ কী? $d[u]$ এর থেকে ডিসকভারি টাইম ছোটো একমাত্র $ancestor(u)$ সেটের নোডগুলোর। $subtree(v)$ এর কোনো নোড ব্যাকএজ দিয়ে $ancestor(u)$ এর সঙ্গে যুক্ত, সেজন্য $low[v]$ এর মান $d[u]$ এর থেকে কমে গিয়েছে। যদি $d[u] \leq low[v]$ হয়, তাহলেই শুধুমাত্র নোড u একটি আর্টিকুলেশন পয়েন্ট হবে।

আগের গ্রাফেই ডিসকভারি টাইমের পাশাপাশি $low[u]$ এর মানগুলোও দেখি (চিত্র ১২.৫)।

Scanned by CamScanner

তাহলে আমরা আর্টিকুলেশন পয়েন্ট বের করার একটি অ্যালগরিদম পেয়ে গিয়েছি। প্রতিটি নোডের জন্য $d[u], low[u]$ বের করতে পারলেই কাজ শেষ। $low[u]$ বের করা কঠিন কিছু নয়, সুডোকোড দেখলেই পরিস্কার হবে।



চিত্র ১২.৫: চিত্র ১২.১ গ্রাফে নির্দিষ্টক্রমে ভিজিট করার পরে $d[\,]$ / $low[\,]$

```
1  time ←  0
2  d[] ← 0
3  low[] ← 0
4
5  procedure FindArticulationPoint(G, u):
6    time ← time+1
7    low[u] ← d[u] ← time
8    visited[u] ← true
9    no_of_children ←  0
10
11   for each edge u to v in G.adjacentEdges(u) do
12     if v = parent[u]
13       Goto line 11
14     end if
15
16     if visited[v] //This is a backedge
17        low[u] ← min(low[u], d[v])
18     end if
19
20     if not visited[v]   //This is a tree edge
```


Scanned by CamScanner

```
21          parent[v] ← u
22          FindArticulationPoint(G, v)
23          low[u] ← min(low[u], low[v])
24
25          if d[u] <= low[v] and u is not root:
26            Articulation_points[u] ← true
27          end if
28
29          no_of_children ← no_of_children + 1
30        end if
31
32        if no_of_children > 1 and u is root:
33          Articulation_points[u] ← true
34        end if
35      end for
36
37      return
```

সুডোকোড ১২.২

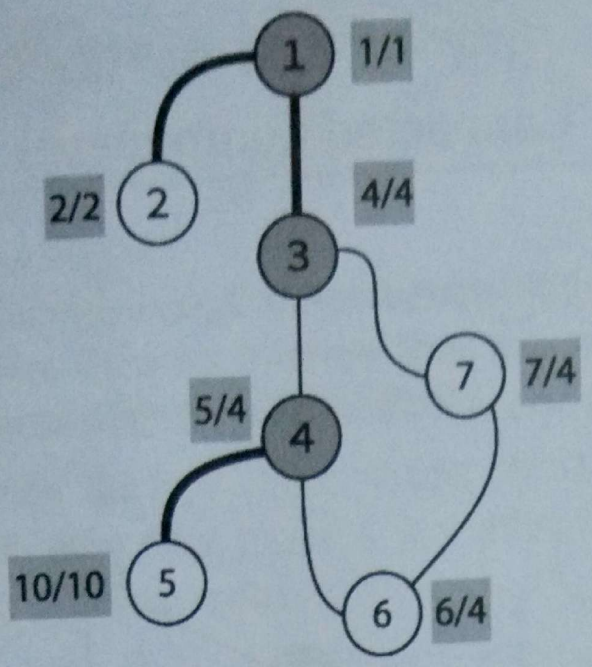
১২.২ – ব্রিজ (Bridge)

ব্রিজ হলো আর্টিকুলেশন পয়েন্টের মতোই একটি বিষয়। গ্রাফ থেকে যে এজ তুলে দিলে গ্রাফটি একাধিক কম্পোনেন্টে ভাগ হয়ে যায় তাকেই বলা হয় ব্রিজ। একে ক্রিটিকাল এজ (Critical Edge) বা কাট এজ (Cut Edge) ও বলা হয়।

উপরের গ্রাফে $[4-5], [1-3]$ আর $[1-2]$ এই তিনটি এজ হলো ব্রিজ। ব্রিজ আর আর্টিকুলেশন পয়েন্টের সুডোকোডের পার্থক্য খালি এক জায়গায় 25 নম্বর লাইনে $d[u] \leq low[v]$ এর জায়গায় $d[u] < low[v]$ লিখতে হবে। এটি কেনো কাজ করে তুমি সহজেই বুঝতে পারবে যদি তুমি সুডোকোডটি বুঝে থাকো, তাই আর ব্যাখ্যা করলাম না।

দুটি নোডের মধ্যে একাধিক এজ থাকলে অবশ্য এটি কাজ করবে না। তখন কী করতে হবে সেটি চিন্তা করা তোমার কাজ!

Scanned by CamScanner



চিত্র ১২.৬: গাঢ় রঙের এজগুলো এই গ্রাফের ব্রিজ

ইউনিক পাথ প্রবলেম

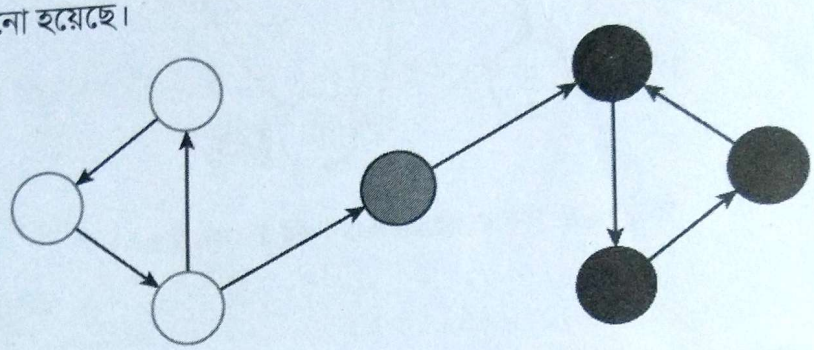
তোমাকে একটি আনডিরেক্টেড গ্রাফ দেওয়া আছে। এরপর তোমাকে Qটি প্রশ্ন করা হলো। প্রতিটি প্রশ্ন হলো এরকম "গ্রাফে নোড u থেকে নোড v তে যাওয়ার জন্য ভিন্ন ভিন্ন পথের সংখ্যা এক নাকি একাধিক?" অর্থাৎ নোড u থেকে নোড v তে যাবার পথটি ইউনিক (Unique) কি না সেটি তোমাকে বলতে হবে। দুটি পথ ভিন্ন হবে যদি পথ দুটিতে অন্তত একটি ভিন্ন এজ থাকে।

ব্রিজ ব্যবহার করে এই সমস্যাটি সমাধান করা যায়। লক্ষ কর নোড u থেকে নোড v তে যাবার পথটি যদি ইউনিক হয়, তাহলে পথের প্রতিটি এজ অবশ্যই একটি করে ব্রিজ হবে। তাহলে তোমার কাজ হবে শুরুতেই গ্রাফের ব্রিজগুলো রেখে বাকি এজগুলো ফেলে দেওয়া। এবার যদি u আর v একই সাবগ্রাফে বা কম্পোনেন্টে থাকে তাহলে তাদের মধ্যকার পথ অবশ্যই ইউনিক।

Scanned by CamScanner

অধ্যায় ১৩ – স্ট্রংলি কানেক্টেড কম্পোনেন্ট (Strongly Connected Component)

একটি ডিরেক্টেড গ্রাফের স্ট্রংলি কানেক্টেড কম্পোনেন্ট (Strongly Connected Component) বা এসসিসি (SCC) হলো এমন একটি কম্পোনেন্ট যার প্রতিটি নোড থেকে অন্য সব নোডে যাওয়ার পথ আছে। চিত্র ১৩.১ এ একটি গ্রাফের প্রতিটি স্ট্রংলি কানেক্টেড কম্পোনেন্ট আলাদা রঙ দিয়ে দেখানো হয়েছে।



চিত্র ১৩.১: একটি ডিরেক্টেড গ্রাফের বিভিন্ন স্ট্রংলি কানেক্টেড কম্পোনেন্ট

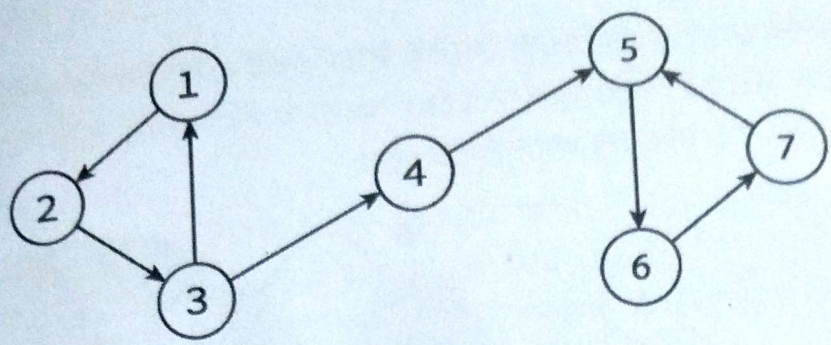
ডেপথ ফার্স্ট সার্চের ফিনিশিং টাইমের ধারণা ব্যবহার করে আমরা $O(V+E)$ তে একটি গ্রাফের স্ট্রংলি কানেক্টেড কম্পোনেন্টগুলোকে আলাদা করে ফেলতে পারি। এই লেখাটি পড়ার আগে অবশ্যই টপোলজিকাল সর্টিং এবং ডেপথ ফার্স্ট সার্চের ডিসকভারি এবং ফিনিশিং টাইম সম্পর্কে ধারণা থাকতে হবে।

চিত্র ১৩.২ এর গ্রাফটি দেখ।

প্রথমেই একটি ভুল পদ্ধতিতে অনেকে এসসিসি বের করার চেষ্টা করে। সেটি হলো যেকোনো নোড থেকে ডিএফএস চালিয়ে যেসব নোডে যাওয়া যায় তাদেরকে একটি কম্পোনেন্ট হিসেবে ধরা। কিন্তু খুব সহজেই বোঝা যায় এটি কাজ করবে না, চিত্র ১৩.২ এর গ্রাফে নোড 1 থেকে ডিএফএস চালালে সবগুলো নোড ভিজিট করা যাবে, কিন্তু নোড 1 থেকে নোড 4 এ যাওয়া গেলেও নোড 4 থেকে নোড 1 এ যাওয়ার কোনো পথ নেই, তাই এরা একই কানেক্টেড কম্পোনেন্টের অংশ নয়। এই পদ্ধতিতে সমস্যা হলো ডিএফএস কানেক্টেড কম্পোনেন্ট থেকে বের হয়ে অন্য কম্পোনেন্টে চলে যায়। এই সমস্যা সমাধান করতে আমরা একটু বুদ্ধিমানের মতো ডিএফএস চালাব।

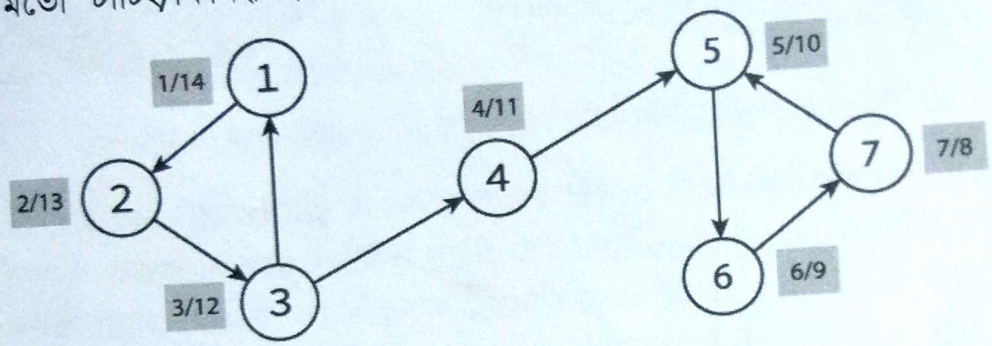
দুটি নোড u, v একই এসসিসিতে থাকবে শুধুমাত্র যদি u থেকে v তে যাবার পথ থাকে এবং v থেকে u তে যাওয়ারও পথ থাকে।

Scanned by CamScanner



চিত্র ১৩.২: একটি ডিরেক্টেড গ্রাফ

প্রথমে আমরা নোড 1 থেকে ডিএফএস চালিয়ে সবগুলো নোডের ডিসকভারি টাইম আর ফিনিশিং টাইম লিখে ফেলি। নোডগুলো 1, 2, 3, 4, 5, 6, 7 অর্ডারে ভিজিট করলে আমরা চিত্র ১৩.৩ এর মতো স্টার্টিং/ফিনিংশিং টাইম পাবো।



চিত্র ১৩.৩: নির্দিষ্টক্রমে ভিজিট করার পর ডিসকভারি টাইম / ফিনিশিং টাইম

এখন বোঝার সুবিধার জন্য স্ট্রংলি কানেক্টেড কম্পোনেন্টের সবগুলো নোডকে একটি বড় নোড মনে করি (চিত্র ১৩.৪)।

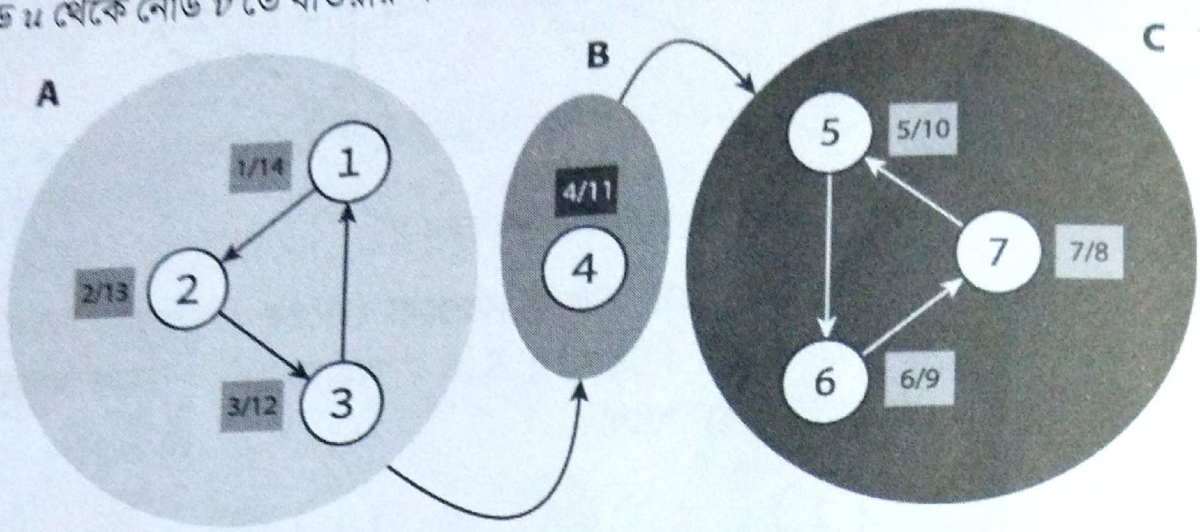
লক্ষ কর, গ্রাফটিকে এভাবে "ডিকম্পোজিং" করার পর গ্রাফটিতে আর কোনো সাইকেল থাকা সম্ভব নয়, অর্থাৎ গ্রাফটি একটি ড্যাগ (DAG) বা ডিরেক্টেড অ্যাসাইক্লিক গ্রাফে পরিণত হয়েছে। এখন বড় নোডগুলোকে সহজেই টপোলজিকাল অর্ডারে সাজানো সম্ভব, অর্ডারটি হবে A, B, C।

এখন আবার লক্ষ কর ড্যাগে একটি নোড $D1$ থেকে অন্য নোড $D2$ তে যদি যাওয়া যায় তাহলে $D1$ টপোলজিকাল অর্ডারে অবশ্যই $D2$ এর আগে থাকবে। আবার আমরা আগেই জানি যে টপোলজিকাল অর্ডারে যে আগে থাকে তার ফিনিশিং টাইম বেশি হয় কারণ অন্যান্য সব নোডের কাজ শেষ করে ওই নোডে ফিরে আসতে হয়।

তাহলে $D1$ যদি টপোলজিকাল অর্ডারে $D2$ এর আগে থাকে তাহলে যেসব ছোট ছোট নোড নিয়ে $D2$ গঠিত হয়েছে তাদের সবার ফিনিংশিং টাইম অবশ্যই $D1$ এর সব নোডের থেকে কম হবে।

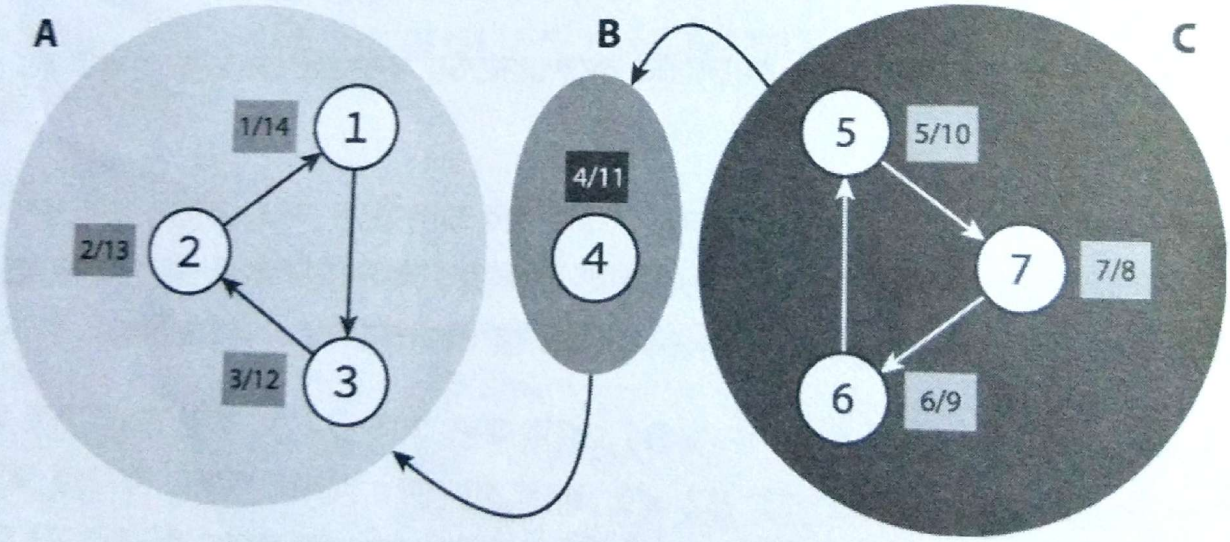
Scanned by CamScanner

এখন নোড u থেকে নোড v তে যাওয়া গেলেই তারা একই এসসিসিয়ের অন্তর্ভূক্ত নয়, নোড v থেকে নোড u তে যাওয়ার পথও থাকতে হবে। অথবা আমরা বলতে পারি "উল্টো-গ্রাফ" এও নোড u থেকে নোড v তে যাওয়ার পথ থাকতে হবে!



চিত্র ১৩.৪: প্রতিটি স্ট্রংলি কানেক্টেড কম্পোনেন্টকে একটি করে বড় মনে করি

যদি গ্রাফের এজগুলোর চিহ্ন উল্টে দেওয়া হয়, তাহলেও এসসিসিগুলো একই থাকে। একে বলা হয় ট্রান্সপোজ (Transpose) গ্রাফ, ট্রান্সপোজ গ্রাফে সাইকেলগুলোর কোনো পরিবর্তন হয় না। মূল গ্রাফে যদি নোড u, v একই এসসিসিয়ের মধ্যে থাকে তাহলে তারা অবশ্যই একই সাইকেলের মধ্যে থাকবে। এটাই আমাদের অ্যালগরিদমের মূল ভিত্তি।



চিত্র ১৩.৫: এজগুলোকে উল্টে দেওয়া হয়েছে

চিত্র ১৩.৫ এ আগের গ্রাফের এজগুলো উল্টে দেওয়া হয়েছে। ডিসকভারি এবং ফিনিশিং টাইম আগেরটাই লেখা আছে।

Scanned by CamScanner

এখন লক্ষ কর তুমি যদি শুরুতে টপোলজিকাল অর্ডারে আগে থাকা নোডগুলো থেকে ডিএফএস চালাও অর্থাৎ যার ফিনিশিং টাইম বড় সেখান থেকে শুরু কর তাহলে তুমি প্রথম এসসিসিটি পেয়ে যাবে।

উপরের গ্রাফে নোড 1 এর ফিনিশিং টাইম সবথেকে বেশি (14)। নোড 1 থেকে ডিএফএস চালালে তুমি যেতে পারবে $\{1,2,3\}$ নোডগুলোতে যারা একই এসসিসিয়ের অংশ। এবার $\{1,2,3\}$ নোডগুলোকে গ্রাফ থেকে মুছে ফেল। এরপর নোড 4 এর ফিনিশিং টাইম বড়। নোড 4 থেকে শুধুমাত্র $\{4\}$ এ যাওয়া যায়। এরপর নোড 5 থেকে ডিএফএস চালাব, সেখান থেকে যাওয়া যায় $\{5,6,7\}$ নোডগুলোতে যারা একটি এসসিসিয়ের অংশ।

যে নোডগুলো একই কম্পোনেন্টের অংশ তাদেরকে আমরা আলাদা লিস্টে সেভ করে রাখব নিচের সুডোকোডটি দেখো,

```
procedure DFS(G, u):
  color[u]  ← GREY
  for all edges from u to v in G.adjacentEdges(u) do
    if color[v] = WHITE
      DFS(G,v)
    end if
  end for
  stk.add(source)
  return

procedure DFS2(R,u, mark):
  components[mark].add(u) //save nodes of the new component
  visited[u] ← true

  for all edges from u to v in R.adjacentEdges(u) do
    if visited[u] ← false
      DFS2(R,v, mark)
    end if
  end for

  return

procedure findSCC(G):
  stk ← an empty stack
  visited[] ← false
  color[] ← WHITE
  components[] ← null
```


Scanned by CamScanner

```
28    mark ← 0
29
30    for each node u in G do
31      if color[u] = WHITE
32        DFS(G,u)
33      end if
34    end for
35
36    R ← reverseEdges(G)
37
38    while stk not empty
39        u ← stk.removeTop()
40        if visited[u] = false
41          mark ← mark + 1 //A new component found
42          DFS2(R,u,mark)
43        end if
44    end while
45
46    return components
```

সুডোকোড ১৩.১

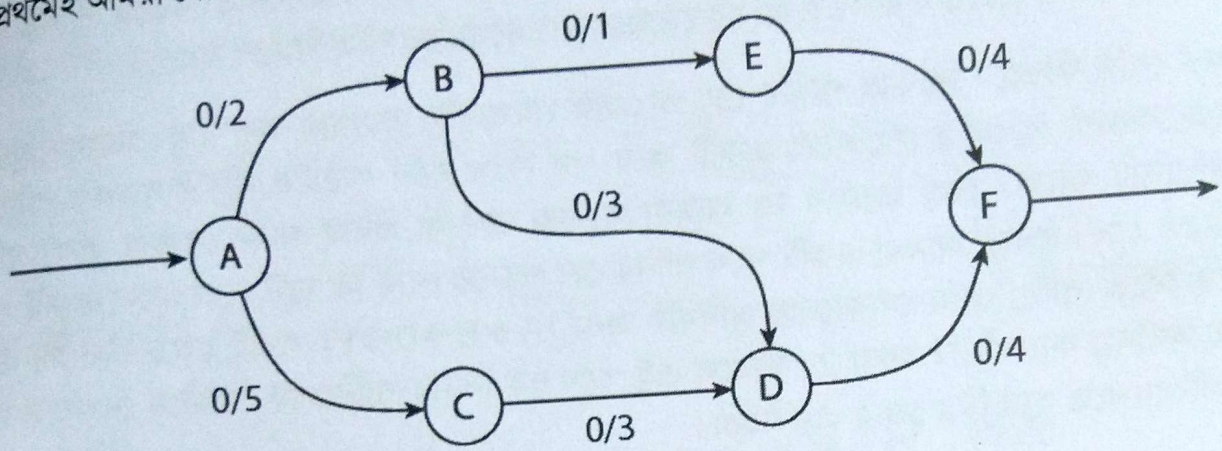
কোডটি একটু বড় মনে হলেও বোঝা খুব সহজ। প্রথমে একটি ডিএফএস চালিয়ে ফিনিশিং টাইম অনুযায়ী নোডগুলো সর্ট করছি। একটি স্ট্যাক ব্যবহার করে কাজটি করছি। যার ফিনিশিং টাইম কম সে কাজ আগে শেষ করে 12 নম্বর লাইনে আসবে, তখন সেই নোডটি স্ট্যাকে ঢুকিয়ে রাখব। সবশেষে স্ট্যাকের উপরে যে নোড থাকবে তার ফিনিশিং টাইম হবে সবচেয়ে বেশি। এরপর দ্বিতীয় ডিএফএস চালিয়ে কম্পোনেন্টগুলোকে আলাদা করে ফেলছি। $mark$ নামের ভ্যারিয়েবলটি ব্যবহার করছি প্রতিটি কম্পোনেন্টের আলাদা নাম দেওয়ার জন্য, ছবিতে যেভাবে A, B, C নাম দেওয়া হয়েছে।

Scanned by CamScanner

অধ্যায় ১৪ – ম্যাক্সিমাম ফ্লো (Maximum Flow)

ম্যাক্সিমাম ফ্লো (Maximum Flow) বা ম্যাক্সফ্লো (Maxflow) ব্যবহার করে অসংখ্য ধরনের গ্রাফ সমস্যার সমাধান করা যায়, এই বিষয়টি খুবই গুরুত্বপূর্ণ। সোভিয়েত ইউনিয়নে রেললাইনের ডিজাইন করতে গিয়ে প্রথম এই সমস্যা সমাধানের মডেল তৈরি করা হয়।

প্রথমেই আমরা দেখি একটি খুবই সাধারণ ম্যাক্সফ্লো সমস্যা।



চিত্র ১৪.১: একটি ডিরেক্টেড গ্রাফ

তোমাকে চিত্র ১৪.১ এর মতো একটি গ্রাফ দেওয়া আছে। মনে কর গ্রাফের প্রতিটি এজ একটি করে পানির পাইপ। প্রতিটি পাইপ দিয়ে প্রতি সেকেন্ডে কত লিটার পানি প্রবাহিত হতে পারবে সেটির একটি সীমা আছে যেটিকে বলা হয় পাইপের **ক্যাপাসিটি (Capacity)**। আর কোনো পাইপ দিয়ে সেকেন্ডে যতটুকু পানি যাচ্ছে সেটি হলো পাইপটির **ফ্লো (Flow)**।

প্রতিটি এজের সঙ্গে "ফ্লো/ক্যাপাসিটি" উল্লেখ করে দেওয়া হয়েছে। যেমন [A→B] এজ দিয়ে সেকেন্ডে সর্বোচ্চ 2 লিটার পানি যেতে পারে এবং এই মুহূর্তে সেকেন্ডে পানি যাচ্ছে 0 লিটার। শুধুমাত্র নোড A দিয়ে পানি প্রবেশ করানো যায় এবং শুধুমাত্র নোড F দিয়ে পানি বের হয়ে যায়। A কে বলা হয় **সোর্স (Source)** এবং F হলো **সিংক (Sink)**। তোমাকে বলতে হবে নোড A থেকে নোড F এ প্রতি সেকেন্ডে সর্বোচ্চ কত লিটার পানি প্রবাহিত করা যাবে? অর্থাৎ পানির "ফ্লো" সর্বোচ্চ কত হতে পারে?

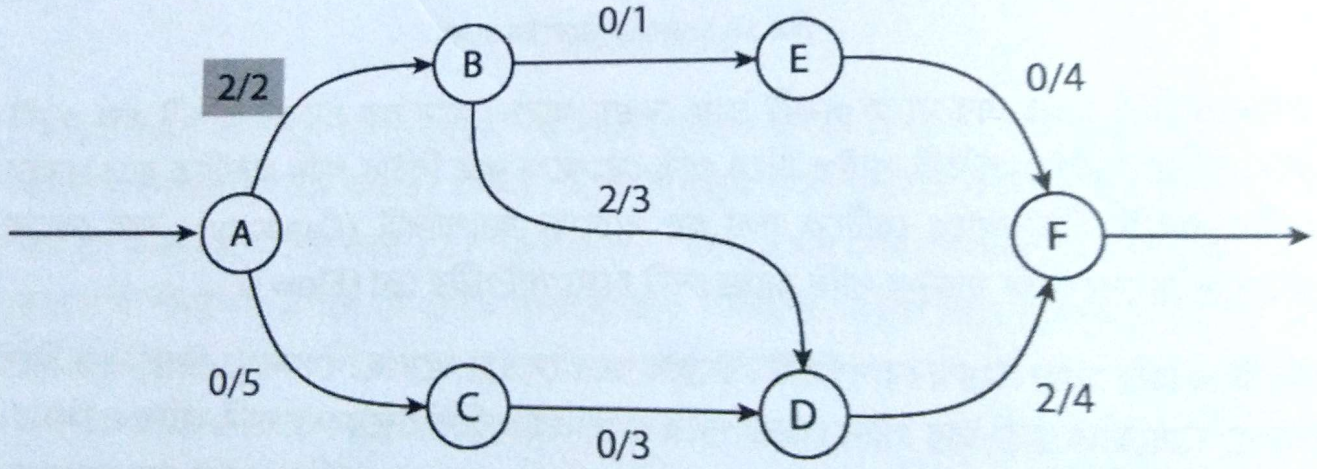
মূল সমস্যাটি সমাধানের আগে আমরা ছোট একটি সমস্যা সমাধান করি। [A→C→D→F] এই পথটি ব্যবহার করে পানি সোর্স থেকে সিংকে গেলে সর্বোচ্চ কত লিটার পানি প্রবাহিত হতে পারবে? ছবিতে দেখা যাচ্ছে [A→C] এজের ক্যাপাসিটি 5, [C→D] এজের ক্যাপাসিটি 3 এবং

Scanned by CamScanner

[D→F] এজের ক্যাপাসিটি 4। এখানে সর্বনিম্ন ক্যাপাসিটি হলো 3, তাই আমরা কোনো সময়ই তিন লিটারের বেশি পানি এই পথে পাঠাতে পারব না।

কোনো পথের সর্বনিম্ন ক্যাপাসিটির এজ সেই পথের ফ্লো নিয়ন্ত্রণ করে। বোতলের সরু মুখের সঙ্গে তুলনা দিয়ে এই ব্যাপারটিকে বলা হয় **বটলনেক (Bottleneck)**। যেমন ধর তোমার বাসায় 5MBps এর ইন্টারনেট সংযোগ আছে, এখন তুমি একটি মুভি ডাউনলোড করতে চাচ্ছ। এখন মুভির সার্ভার থেকে তোমার বাসায় ডেটা প্যাকেট আসার আগে অনেকগুলো নেটওয়ার্ক পার হয়ে আসে, কোনো একটি নেটওয়ার্কে গতি মাত্র 1MBps। এখন নেটওয়ার্কের বাকি অংশের গতি যতই বেশি হোক না কেন, তুমি 1MBps এর বেশি গতিতে ডাউনলোড করতে পারবে না, নেটওয়ার্কের সবচেয়ে ধীরগতির অংশই তোমার ডাউনলোডের গতি নিয়ন্ত্রণ করবে।

এখন দেখি কীভাবে সর্বোচ্চ পানির ফ্লো পাওয়ার সমস্যাটির সমাধান করা যায়। সমাধান খুবই সহজ, আমরা প্রতিবার যেকোনো একটি করে পথ দিয়ে পানি পাঠাতে থাকব যতক্ষণ পাইপে ক্যাপাসিটি থাকে। কিন্তু এভাবে কি আমরা সর্বোচ্চ ফ্লো বা প্রবাহ পাব? সেজন্য একটু বুদ্ধি খাটাতে হবে। প্রথমে আমরা একটি পথে পানির ফ্লো পাঠিয়ে দেখি কি ঘটে। যেকোনো একটি পথ বেছে নিতে পারি, আমি বোঝানোর সুবিধার জন্য [A→B→D→F] পথটি বেছে নিলাম। এই পথে সর্বনিম্ন ক্যাপাসিটি হলো 2। তাহলে এই পথে দুই লিটার পানির ফ্লো পাঠিয়ে দেওয়ার পর গ্রাফটি দেখতে হবে চিত্র ১৪.২ এর মতো।



চিত্র ১৪.২: [A→B→D→F] পথে ফ্লো পাঠানোর পর

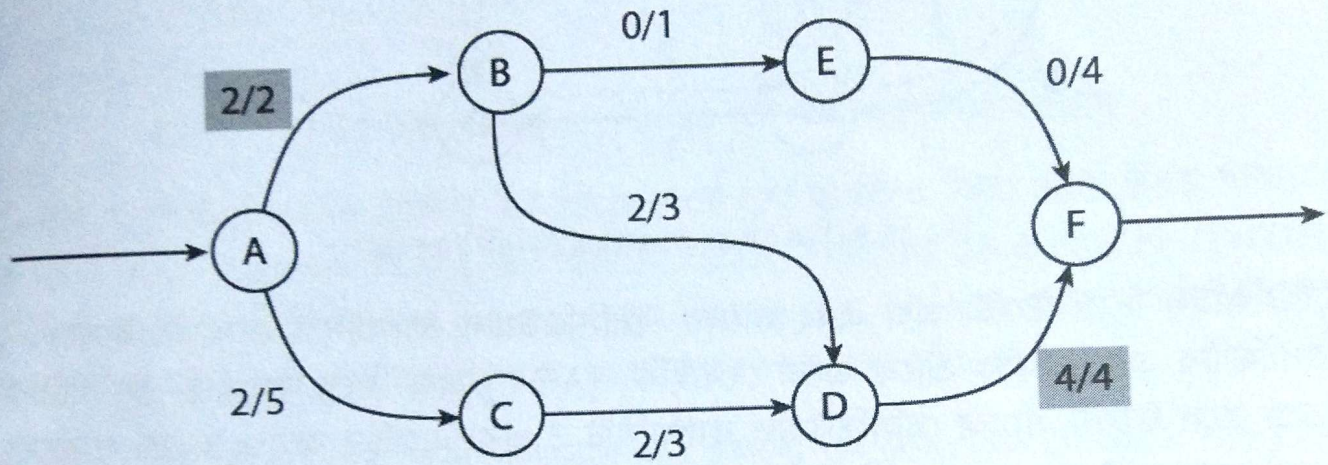
আমরা [A→B→D→F] পথের প্রতিটি এজের ফ্লো দুই বাড়িয়ে দিয়েছি। লক্ষ্য কর [A→B] এজের আসল ক্যাপাসিটি 2 এবং ফ্লো পাঠানো হয়েছে 2, তাই এই এজ দিয়ে আর কোনো ফ্লো পাঠানো যাবে না। তেমনি [B→D] এজের ক্যাপাসিটি 3 কিন্তু ফ্লো পাঠানো হয়েছে 2, তাই এই এজটা দিয়ে আরও 1 (= 3 – 2) ফ্লো পাঠানো সম্ভব। আমরা বলতে পারি [B→D] এজের

Scanned by CamScanner

রেসিডিউয়াল (Residual) ক্যাপাসিটি = 1, Residual শব্দটির অর্থ হলো কোনো কিছু ব্যবহার করে ফেলার পর বেঁচে যাওয়া অংশ।

রেসিডিউয়াল ক্যাপাসিটি = এজের ক্যাপাসিটি – ব্যবহৃত ক্যাপাসিটি বা ফ্লো এর পরিমাণ

এবার একইভাবে [A→C→D→F] পথটি নির্বাচন করি। এই পথে [D→F] এজের আসল ক্যাপাসিটি 4 হলেও 2 ফ্লো আগেই পাঠানো হয়েছে, তাই বর্তমান ক্যাপাসিটি বা রেসিডিউয়াল ক্যাপাসিটি 2 (= 4 – 2)। এই পথটি দিয়ে আরও 2 ফ্লো পাঠানো সম্ভব।



চিত্র ১৪.৩: [A→C→D→F] পথে ফ্লো পাঠানোর পর

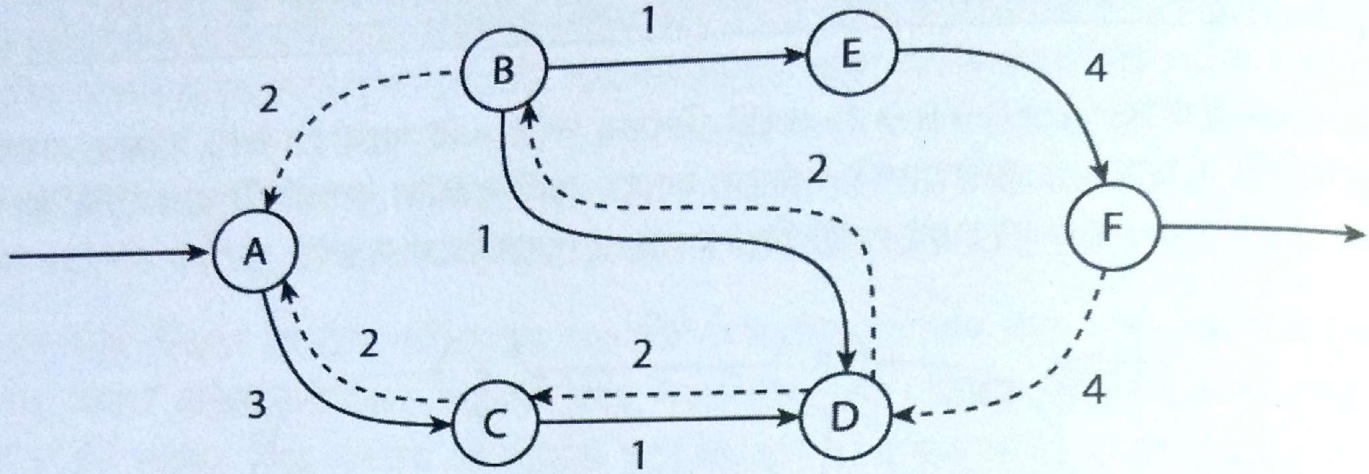
আমরা এখন মোট 2 + 2 = 4 ফ্লো পেলাম, অর্থাৎ পানি প্রবাহের হার এখন সেকেন্ডে 4 লিটার। এই গ্রাফে কি আরও বেশি পানির ফ্লো পাঠানো সম্ভব? [A→B] এবং [D→F] এজ এরই মধ্যে সর্বোচ্চ ক্যাপাসিটিতে পৌঁছে গেছে, এই দুটি এজের কোনোটিকে না নিয়ে সিংকে যাবার পথ গ্রাফে নেই। তাই দেখে মনে হতে পারে আর ফ্লো পাঠানো সম্ভব নয়। কিন্তু আমরা এখন বুদ্ধি খাটিয়ে আরও বেশি ফ্লো পাঠিয়ে দেব!

আমাদের এখন প্রতিটি এজের রেসিডিউয়াল ক্যাপাসিটি ব্যবহার করে রেসিডিউয়াল গ্রাফ আঁকতে হবে কীভাবে আরও ফ্লো পাঠাব সেটি বুঝতে হলে। রেসিডিউয়াল গ্রাফ আকার নিয়ম হলো,

১. গ্রাফের প্রতিটি এজ $[u - v]$ এর ক্যাপাসিটি হবে এজটির রেসিডিউয়াল ক্যাপাসিটির সমান।
২. প্রতিটি এজ $[u - v]$ এর জন্য উল্টো এজ $[v - u]$ এর রেসিডিউয়াল ক্যাপাসিটি হবে $[u - v]$ এজে ফ্লোয়ের সমান।

Scanned by CamScanner

তার মানে কোনো এজ দিয়ে যতটুকু ফ্লো পাঠিয়েছি সেটি হবে উল্টো এজের রেসিডিউয়াল ক্যাপাসিটি। তাহলে রেসিডিউয়াল গ্রাফে মূল এজ এবং উল্টো এজের রেসিডিউয়াল ক্যাপাসিটির যোগফল হবে মূল এজের মোট ক্যাপাসিটির সমান। চিত্র ১৪.৩ এর গ্রাফটির রেসিডিউয়াল গ্রাফটি হবে চিত্র ১৪.৪ এর মতো।



চিত্র ১৪.৪: চিত্র ১৪.৩ এর গ্রাফের রেসিডিউয়াল গ্রাফ

ডটেট লাইন দিয়ে উল্টো এজ এবং তাদের রেসিডিউয়াল ক্যাপাসিটি দেখানো হয়েছে। 0 ক্যাপাসিটির এজগুলোকে গ্রাফে এঁকে দেখাইনি। [A→C] এজ দিয়ে আগে 2 ফ্লো পাঠানো হয়েছে বলে উল্টো এজের রেসিডিউয়াল ক্যাপাসিটি 2 এবং এজটির আরও 3 ফ্লো পাঠানোর ক্ষমতা আছে তাই মূল এজের রেসিডিউয়াল ক্যাপাসিটি 3। ঠিক এভাবে অন্যান্য এজগুলো আঁকা হয়েছে, তুমি নিজে একে যাচাই করে নিতে পারো ঠিকমত বুঝেছো নাকি বা আমার ছবিতে ভুল আছে নাকি।

এখন উল্টো এজে ফ্লো পাঠানোর মানে কি? আমরা তো পাইপের উল্টো দিক দিয়ে পানি পাঠাতে পারবো না বা রাস্তার উল্টো দিক দিয়ে গাড়ি চালাতে পারবো না। **উল্টো দিকে ফ্লো পাঠানোর মানে হলো মূল ফ্লোকে বাতিল করে দেওয়া!** আমরা যদি [D→B] এজ দিয়ে 2 ফ্লো পাঠাই তার মানে [B→D] এজ দিয়ে আসা 2 লিটার পানির প্রবাহকে আমরা বাতিল করে দিচ্ছি। তখন রেসিডিউয়াল গ্রাফে মূল এজের রেসিডিউয়াল ক্যাপাসিটি বেড়ে যাবে এবং উল্টো এজের রেসিডিউয়াল ক্যাপাসিটি কমে যাবে, যোগফল আগের মতই থাকবে!

এখন চিন্তা করে দেখো চিত্র ১৪.৪ দিয়ে আরও ফ্লো সিংকে পাঠানো যায় নাকি। মনে রাখবে চিত্র ১৪.৩ আর চিত্র ১৪.৪ এ একই গ্রাফকেই ভিন্নভাবে আঁকা হয়েছে।

চিত্র ১৪.৪ এ দেখা যাচ্ছে [A→C→D→B→E→F] পথে আরও 1 ফ্লো পাঠানো সম্ভব! এই পথে [D→B] হলো উল্টো এজ। তারমানে [D→B] এজের 1 ফ্লো বাতিল করে দিয়ে আমরা

Scanned by CamScanner