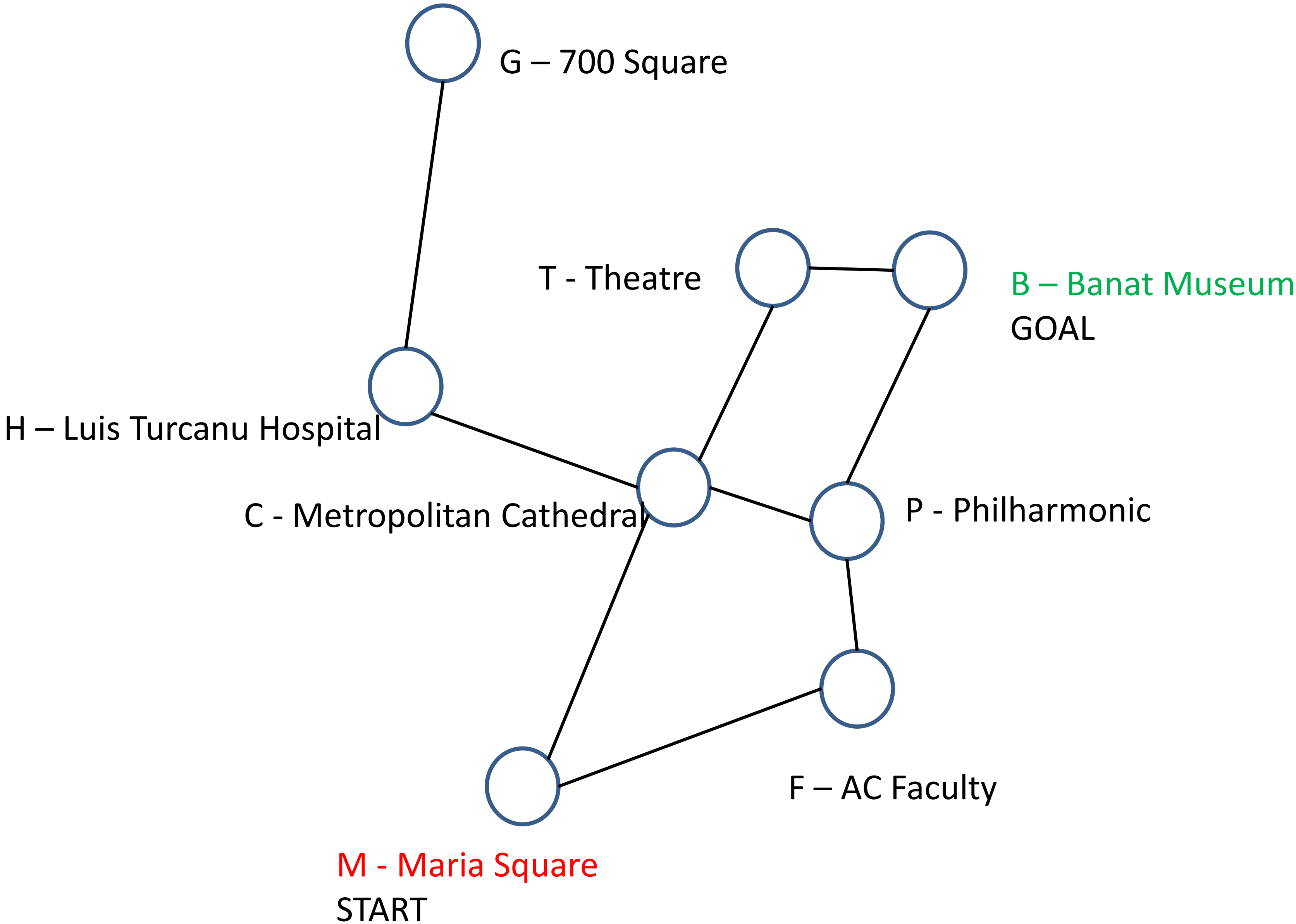
Artificial Intelligence Fundamentals

Search: Depth-First, Hill Climbing, Beam, Optimal, Branch and Bound, A*

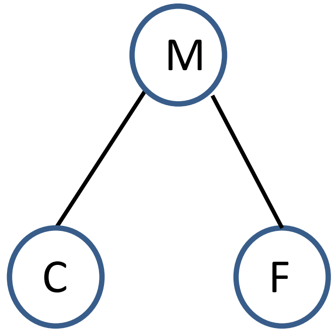
Search - Terminology

- Search algorithm – an algorithms that takes a problem as input and returns a **solution** in the form of an **action sequence**.
- A problem can be defined formally by four components:
  - **Initial state**
  - Possible actions available defined by a **successor function**
  - The **goal test** – determines whether a given state is a goal state
  - A **path cost** – function that assigns a numeric cost to each path.
- Informed vs. Uninformed search – there is some evaluation function that guide your search
- Complete vs. Incomplete search – if there is a solution the algorithm will find it
- Optimal vs. Non-optimal – the solution found is also the best one

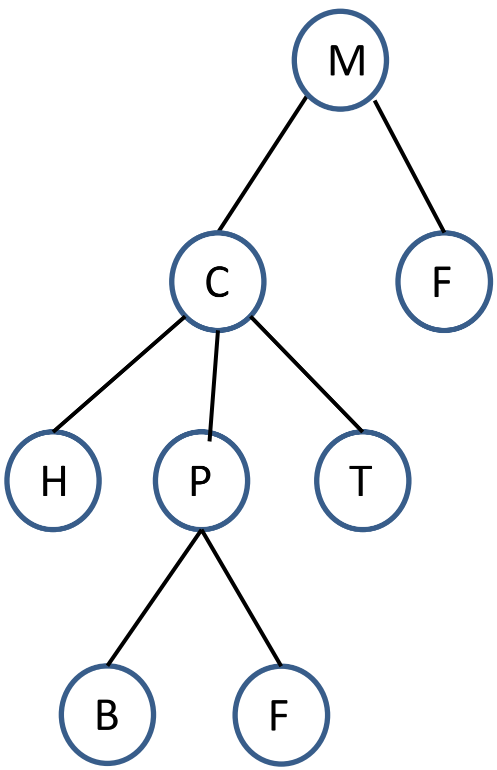
Problem – Navigation in Timisoara



Problem – Navigation in Timisoara



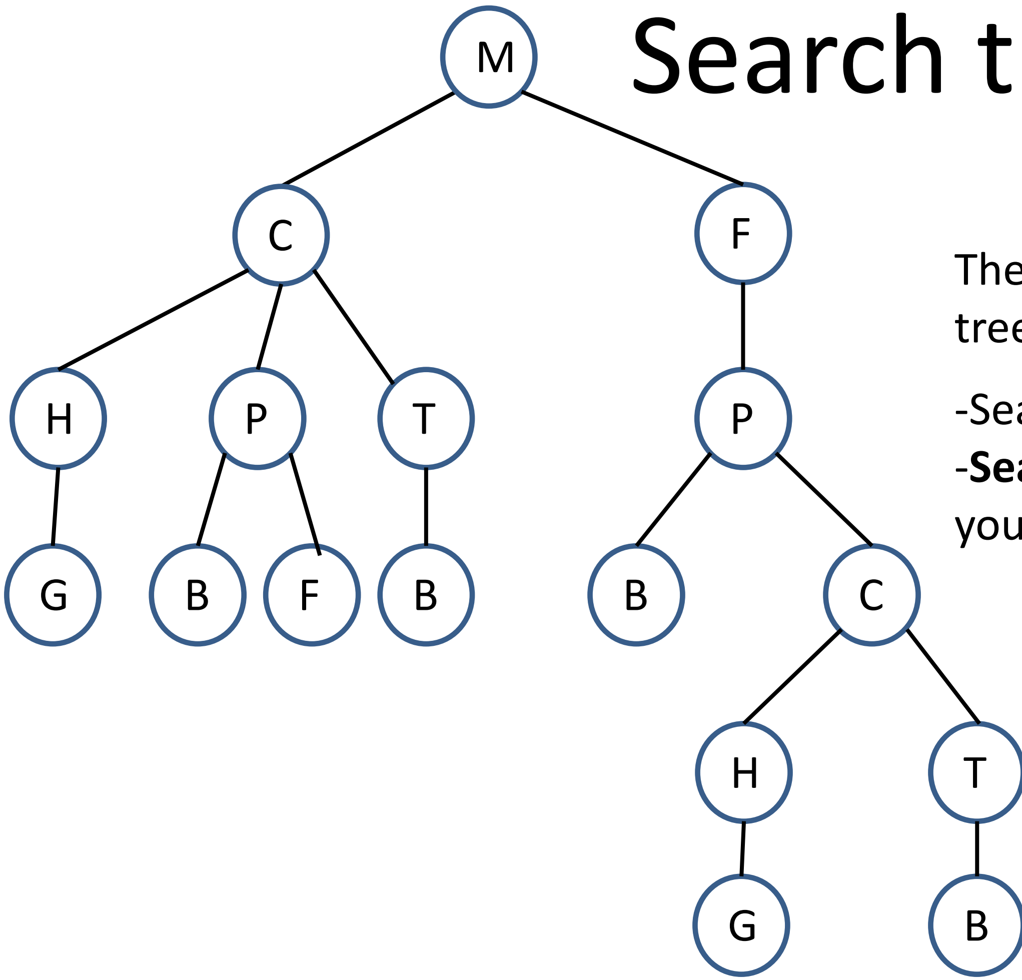
Convention 1 : The nodes appear in lexical order



Convention 2 : When expanding a node don’t put an already visited node as a valid successor. (E.g. M is not a valid successor for C and M is not a valid successor for F)

Problem – Navigation in Timisoara

Search tree



The all possible paths – expansion search tree

-Search is not equal with maps

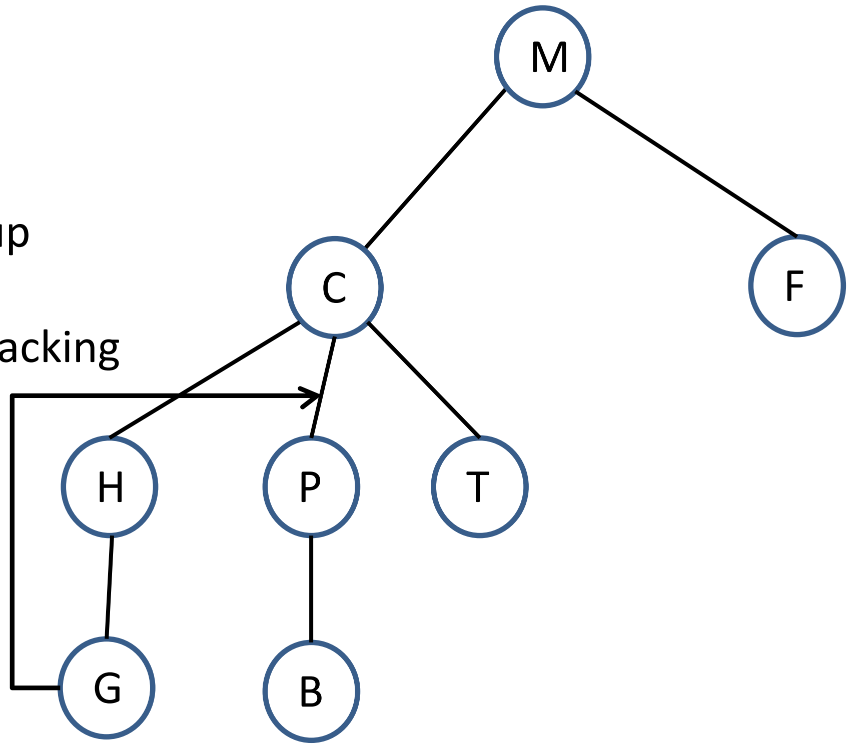
-**Search is about choices** you make when you exploit a map

Depth First Search

- Always expands the **deepest** node
- When all nodes have the same deep, expand the **node from the left** (by convention)
- If a node have no successors, the search **backs-up** to the place we made the last decision and choose another branch
- The process is knows as **back-up** or **backtracking**

Depth First Search

Back-up
or
Backtracking



Depth First Search – The algorithm

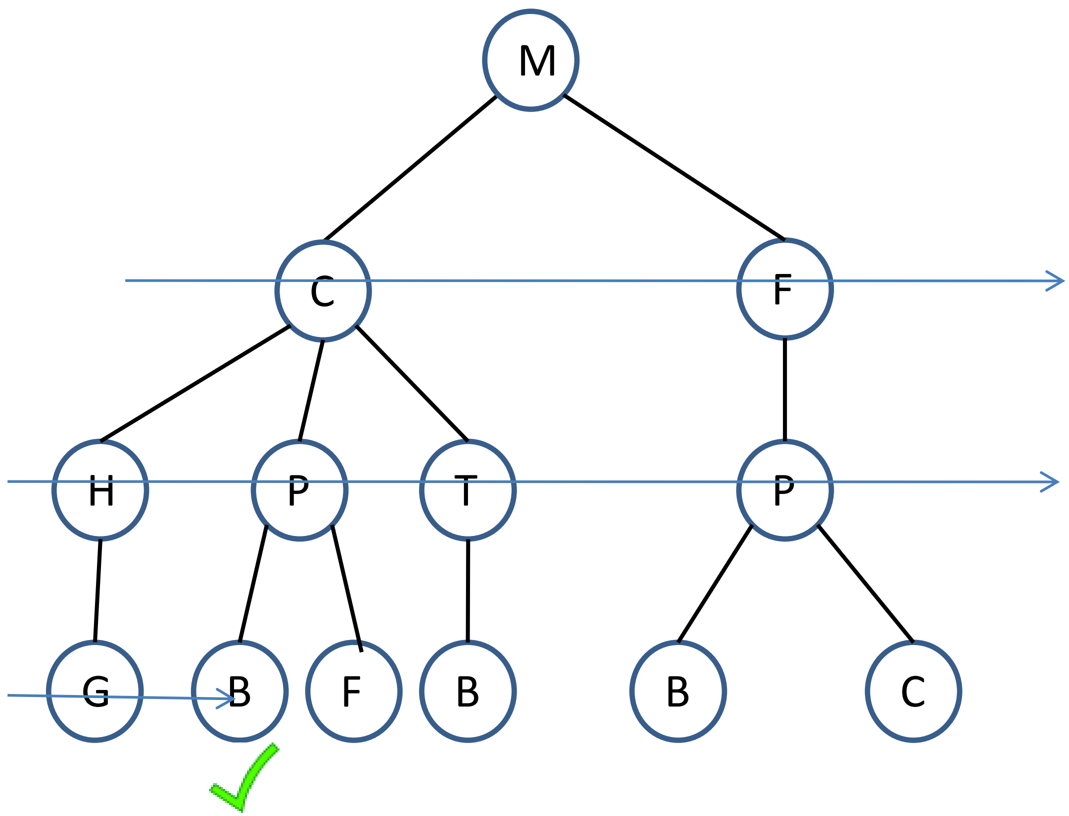
- Initialize the queue with the *Initial state*, having a zero-length path
- **REPEAT** until at least one path in the queue terminates at the goal node OR queue is empty
    - Remove the first path from queue -> *firstPath*
    - Extends the *firstPath* to all the neighbours (except the neighbors that are already in the *firstPath*) of the terminal node -> *newPaths*
    - Add the *newPaths* to the front of the queue
- If the goal is found return SUCCESS

Search characteristics

	Backtracking
Extended search	✗
Depth First	✓

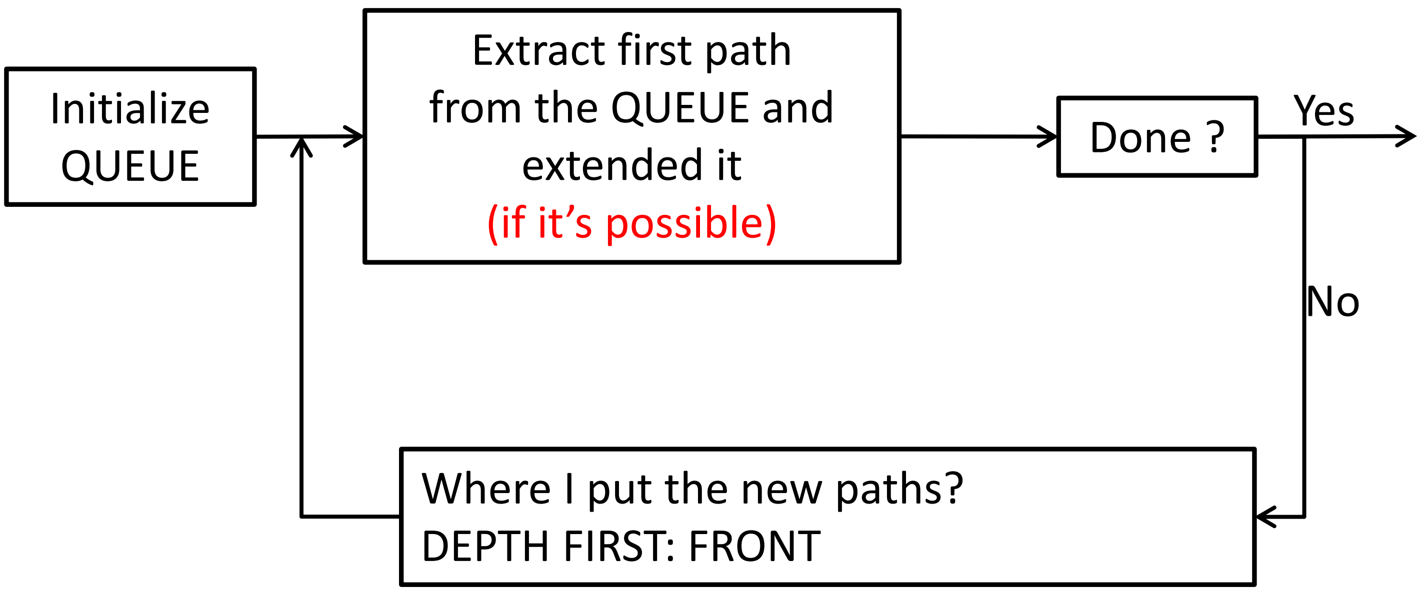
Breadth-first Search

- The **root** node is **expanded** first
- All the node at a **given level are expanded**
- Build up the tree **level by level**



How the search works?

- Helps us to understand the differences between the search algorithms
- Develop a waiting list (**QUEUE**) of paths that are under consideration



Depth First Search

QUEUE
(M)

QUEUE
~~(M)~~
(M C) (M F)

QUEUE
~~(M)~~
~~(M C)~~ (M F)
(M C H) (M C P) (M C T) (M F)

QUEUE
~~(M)~~
~~(M C)~~ (M F)
~~(M C H)~~ (M C P) (M C T) (M F)
(M C H G) (M C P) (M C T) (M F)

QUEUE
~~(M)~~
~~(M C)~~ (M F)
~~(M C H)~~ (M C P) (M C T) (M F)
~~(M C H G)~~ (M C P) (M C T) (M F)
(M C P) (M C T) (M F)

QUEUE
~~(M)~~
~~(M C)~~ (M F)
~~(M C H)~~ (M C P) (M C T) (M F)
~~(M C H G)~~ (M C P) (M C T) (M F)
~~(M C P)~~ (M C T) (M F)
(M C P B) (M C P F) (M C T) (M F)

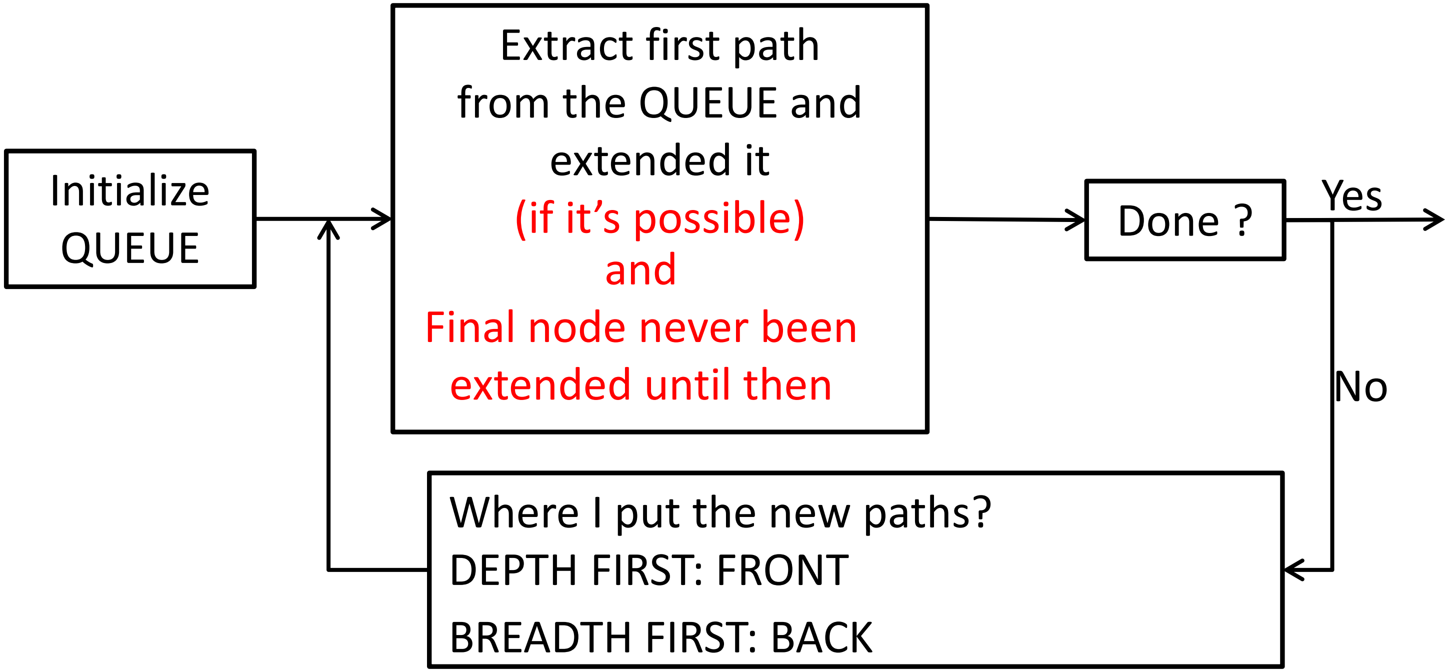
DONE

Breadth First Search – The algorithm

- Initialize the queue with the *Initial state*, having a zero-length path
- **REPEAT** until at least one path in the queue terminates at the goal node OR queue is empty
    - Remove the first path from queue -> *firstPath*
    - Extends the *firstPath* to all the neighbours (except the neighbors that are already in the *firstPath*) of the terminal node -> *newPaths*
    - Add the *newPaths* to the back of the queue
- If the goal is found return SUCCESS

How the search works?

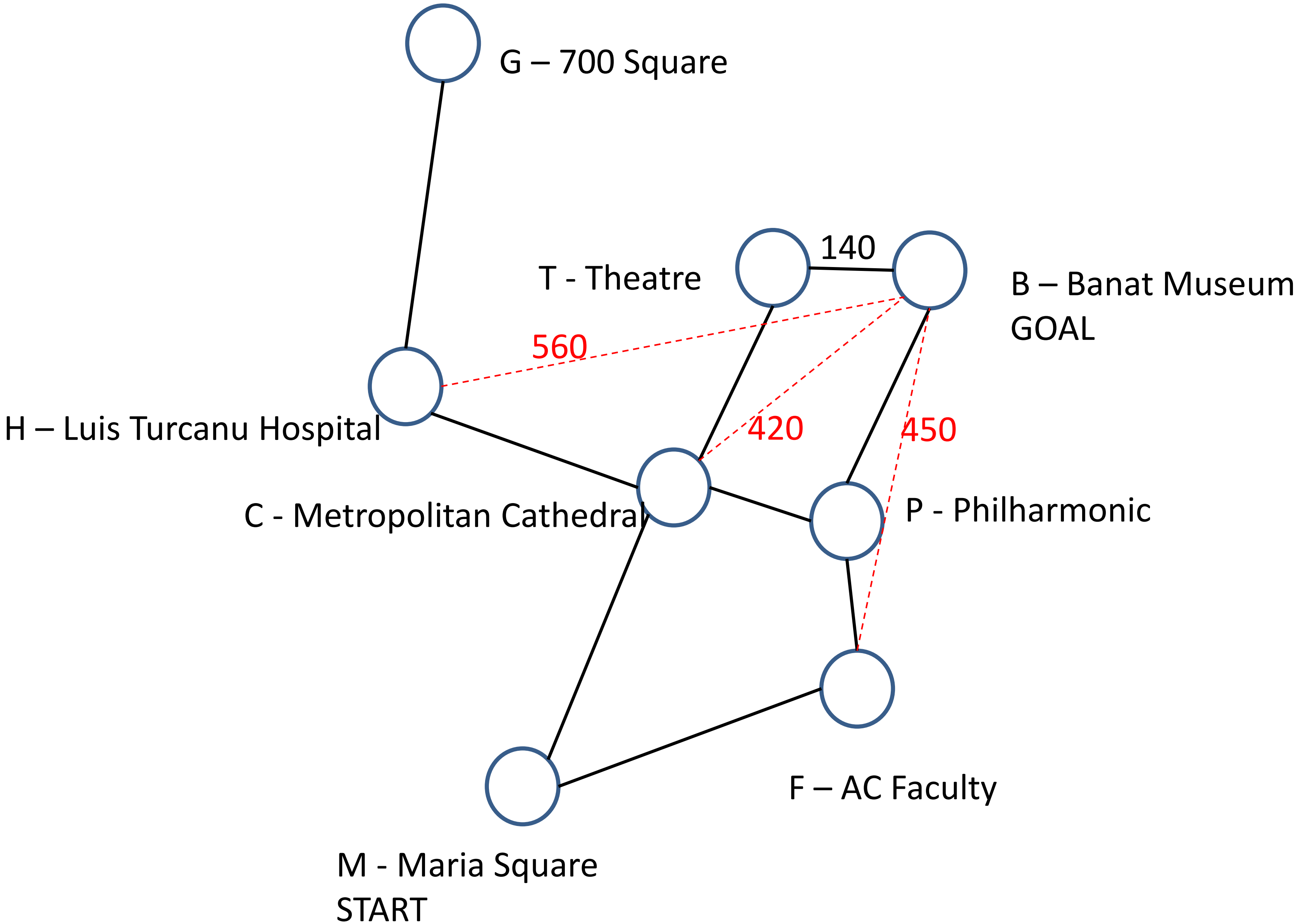
- Helps us to understand the differences between the search algorithms
- Develop a waiting list (**QUEUE**) of paths that are under consideration



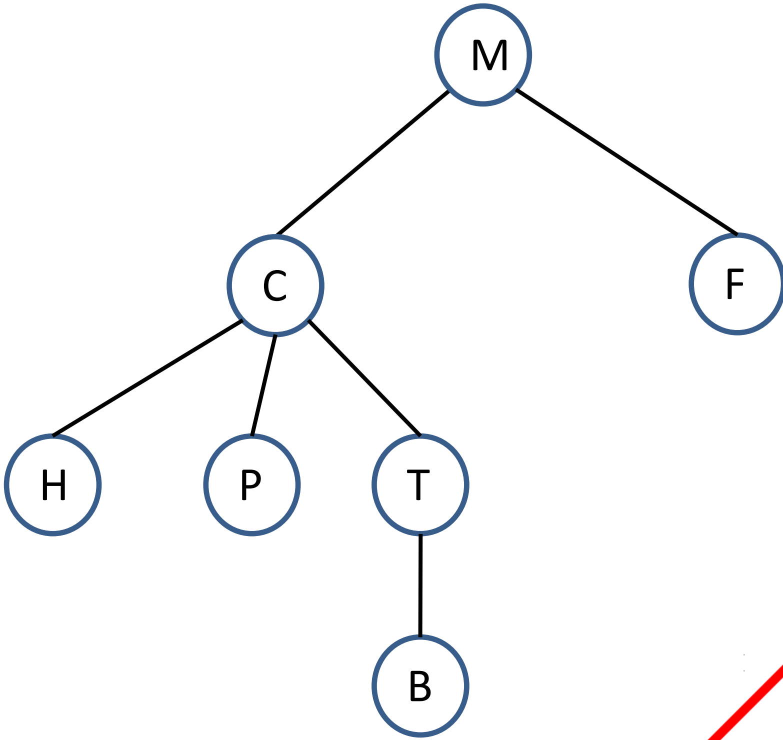
Hill Climbing

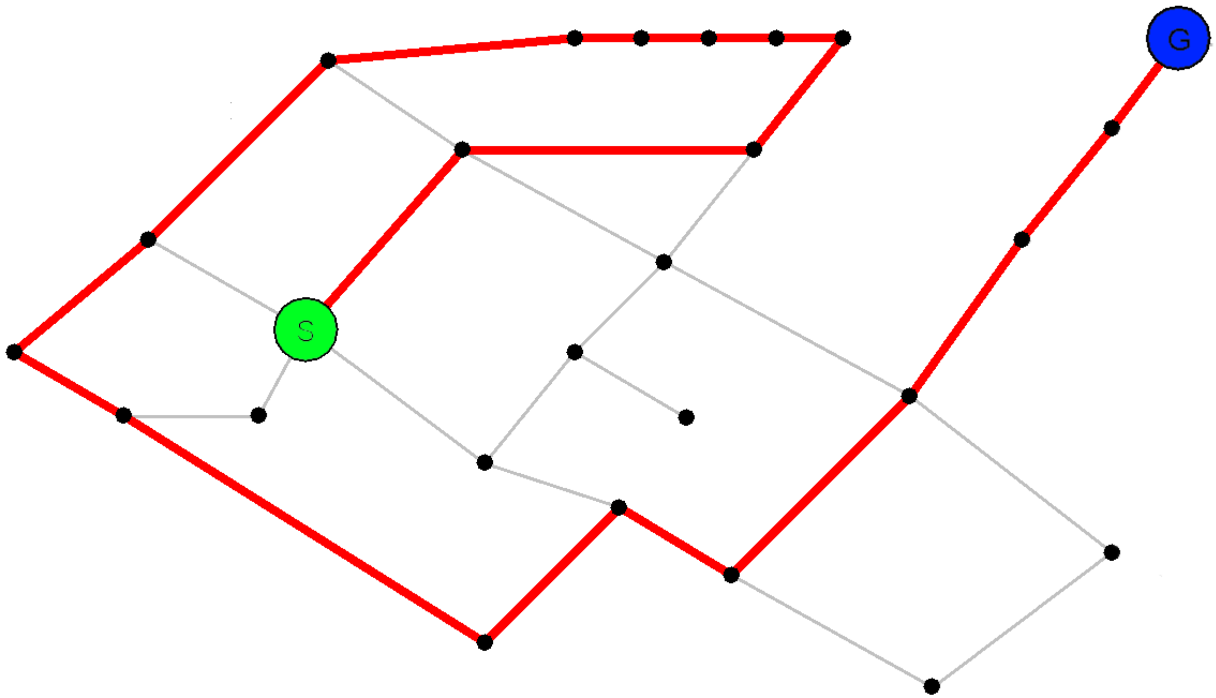
- Depth first search and breadth first search – **uninformed** search strategies
- Incredible **inefficient** in most of the cases
- **Informed(Heuristic)** search
  - uses problem-specific **knowledge ->** create a way to **order the choices**
  - can find solutions more **efficiently**
- Local search algorithms: **best**, **hill-climbing, beam**
- **Hill climbing** (greedy local search)
  - like depth search, but the successors are not listed lexically, they are ordered according to an **objective function**
  - It moves in the direction of **increasing value** (**uphill**)
  - It terminates when it reaches a *peak*
  - Does **not maintain** a search tree

Problem – Navigation in Timisoara



Hill climbing

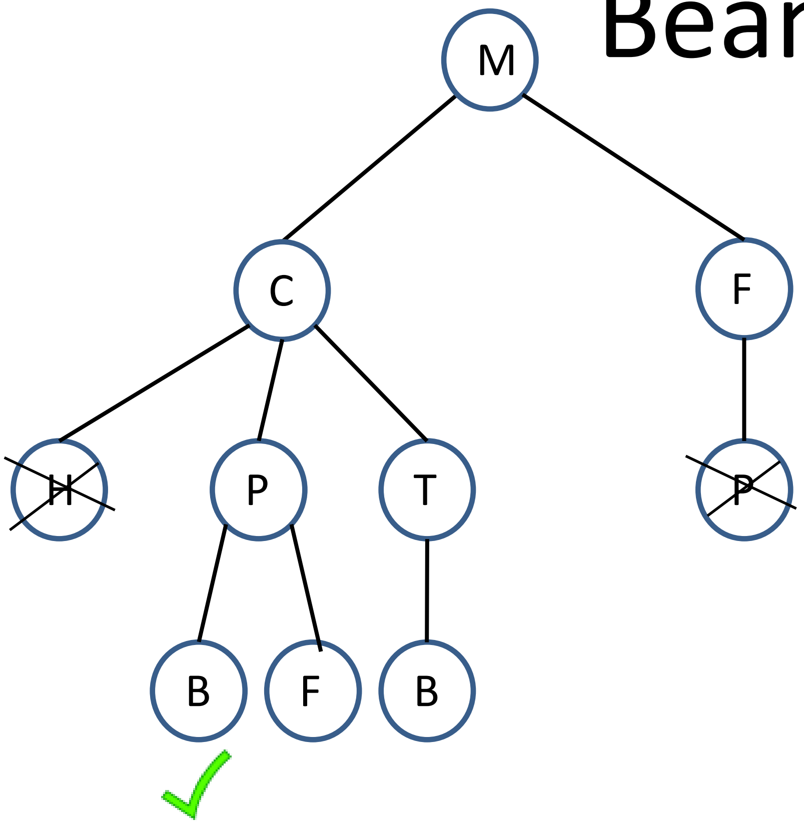




Hill Climbing Search – The algorithm

- Initialize the queue with the *Initial state*, having a zero-length path
- **REPEAT** until at least one path in the queue terminates at the goal node OR queue is empty
    - Remove the first path from queue -> *firstPath*
    - Extends the *firstPath* to all the neighbours (except the neighbors that are already in the *firstPath*) of the terminal node -> *newPaths*
    - Sort the *newPaths* by the estimated distance between the terminates node and the goal
    - Add the new path (if exists) to the front of the queue
- If the goal is found return SUCCESS

Beam search



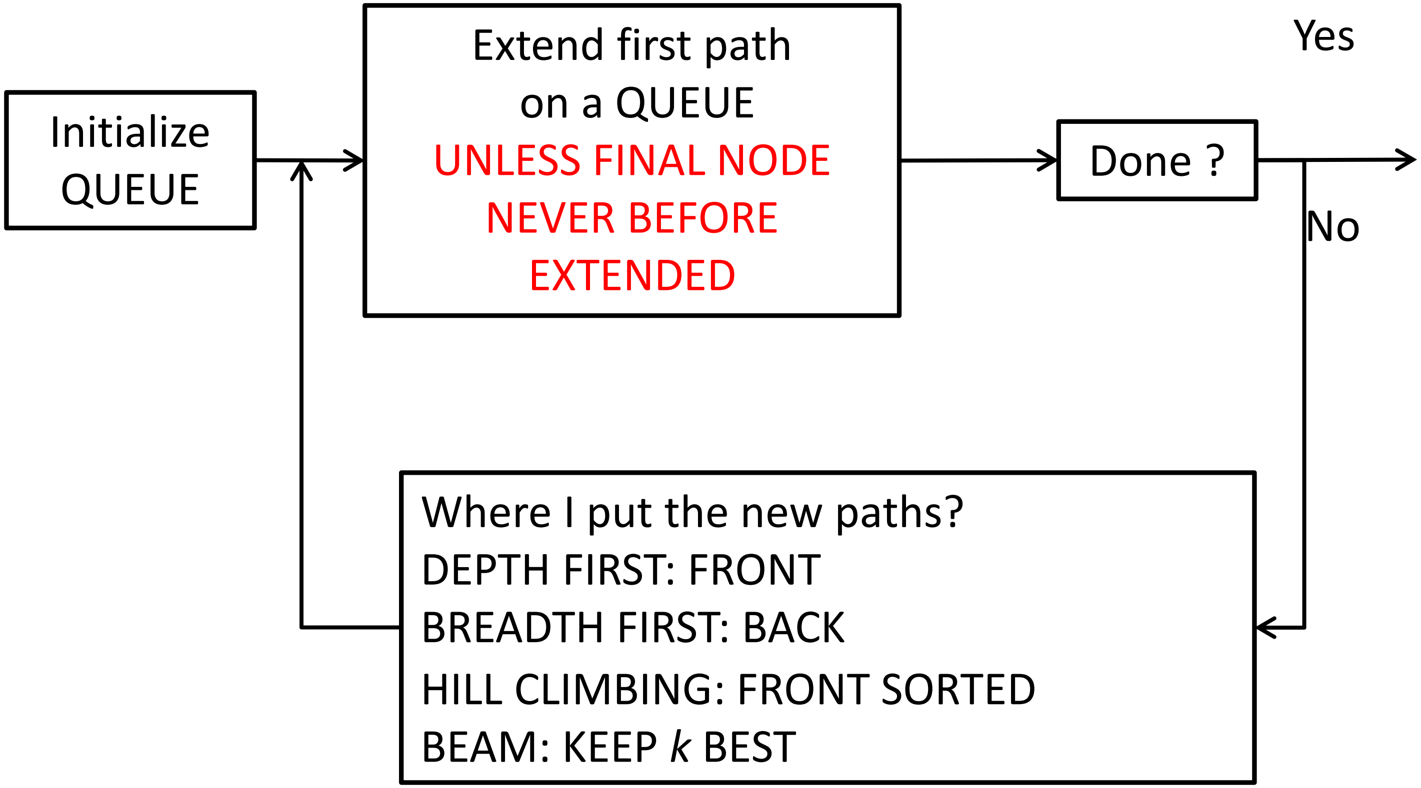
- Like breadth search but keep only a fixed number of possibilities – *k* (usualy small)
- Order the possibilities and take only *k* into consideration
- Example for *k* = 2

Beam Search – The algorithm

- Initialize the queue with the *Initial state*, having a zero-length path
- **REPEAT** until at least one path in the queue terminates at the goal node OR queue is empty
    - Remove the first path from queue -> *firstPath*
    - Extends the *firstPath* to all the neighbours (except the neighbors that are already in the *firstPath*) of the terminal node -> *newPaths*
    - Add the new path (if exists) to the end of the queue
    - If we must go down one level in the search tree, then sort the entire *queue* by the estimated distance between the terminates node and the goal, and keep only first *k* paths
- If the goal is found return SUCCESS

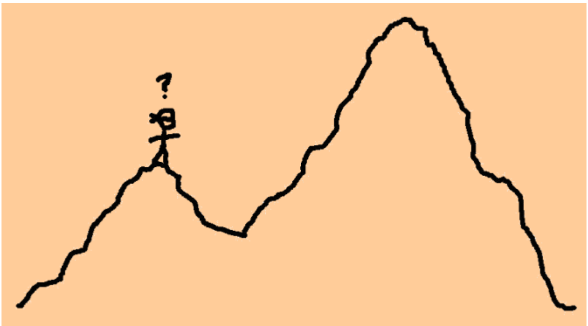
How the search works?

- Helps us to understand the differences between the search algorithms
- Develop a waiting list (**QUEUE**) of paths that are under consideration



Hill climbing problems

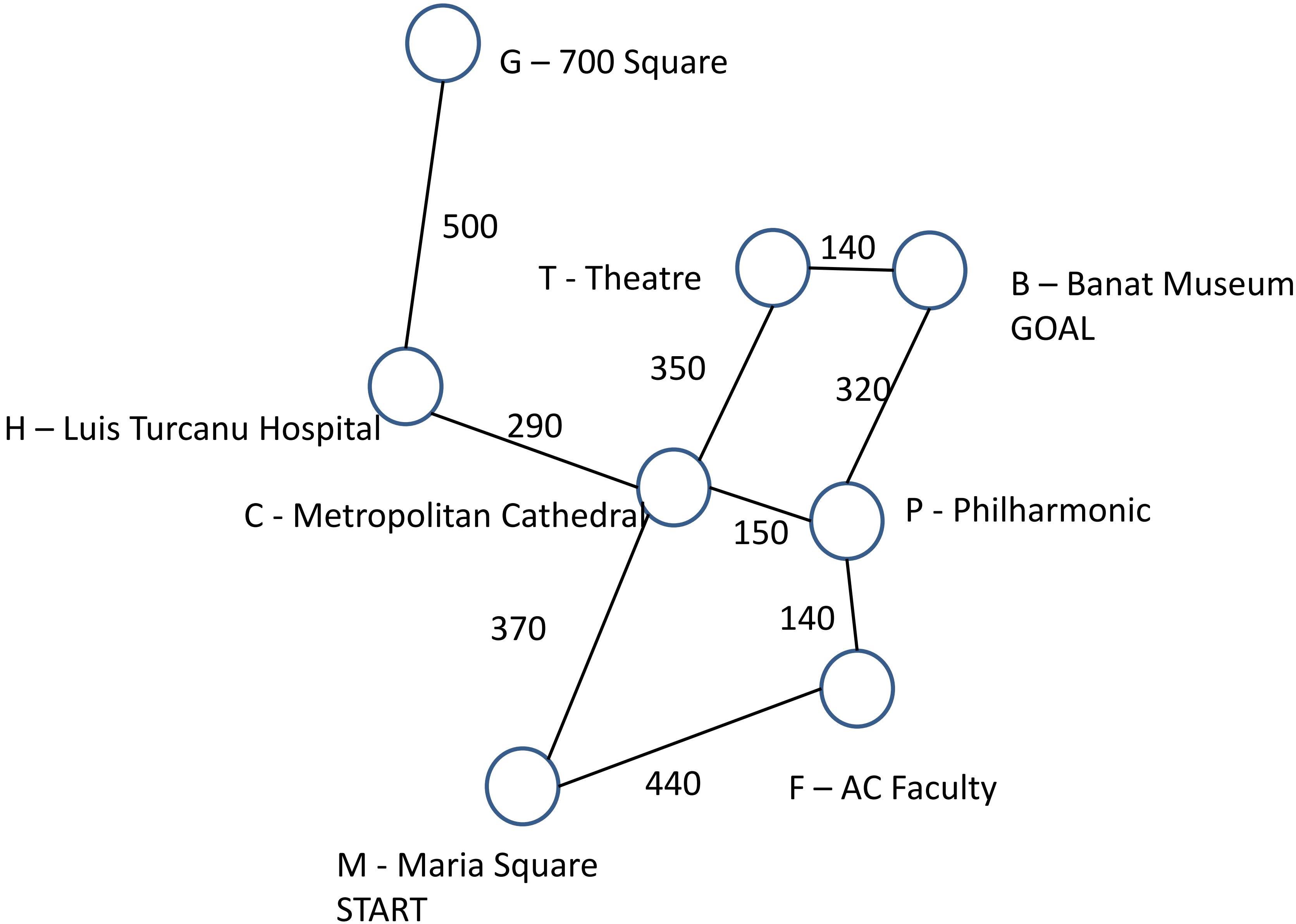
- Problems:
  - Stuck into a local maximum



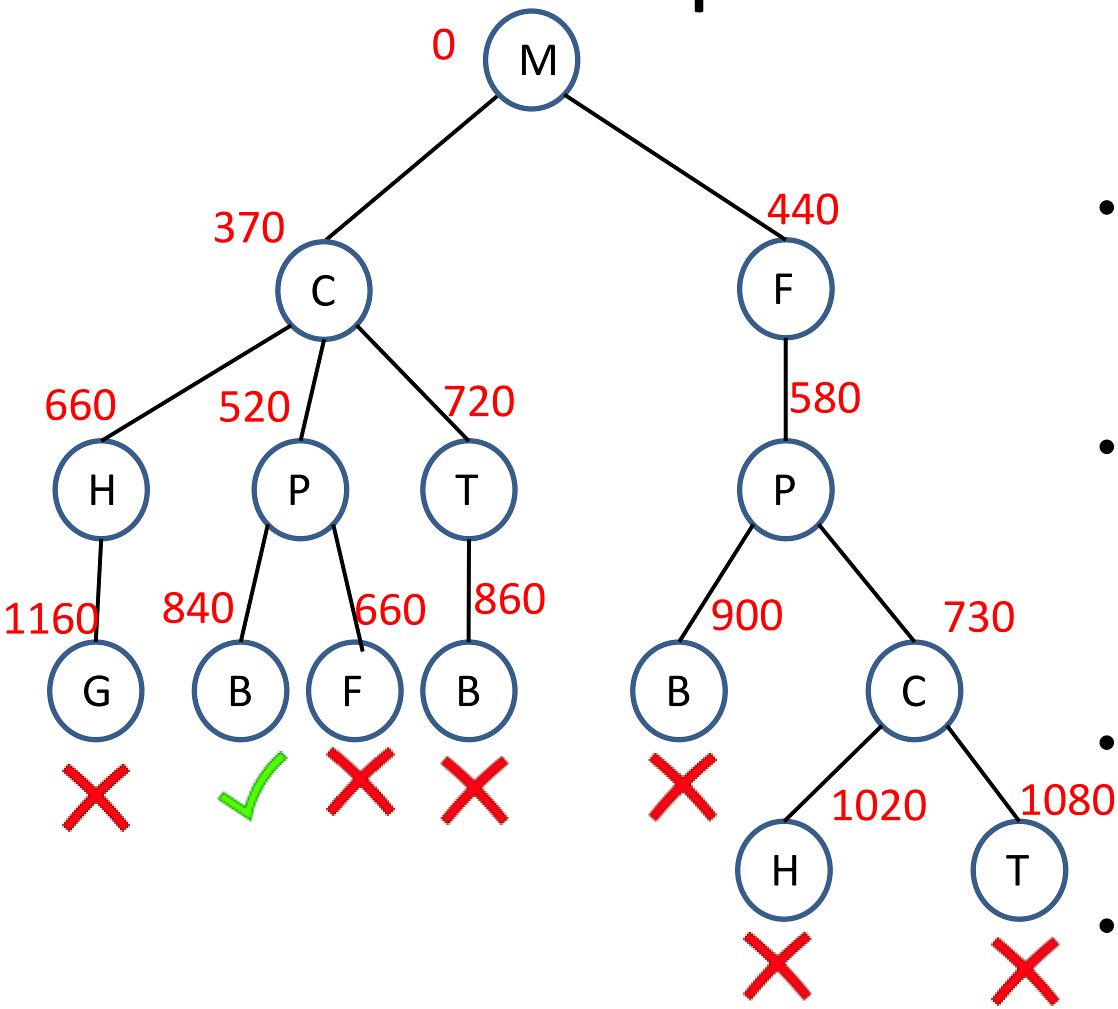
  - Hard to find the *peak* *(plateau problem)*

  - Miss the right direction *(ridge problem)*

Problem – Navigation in Timisoara



Optimal search

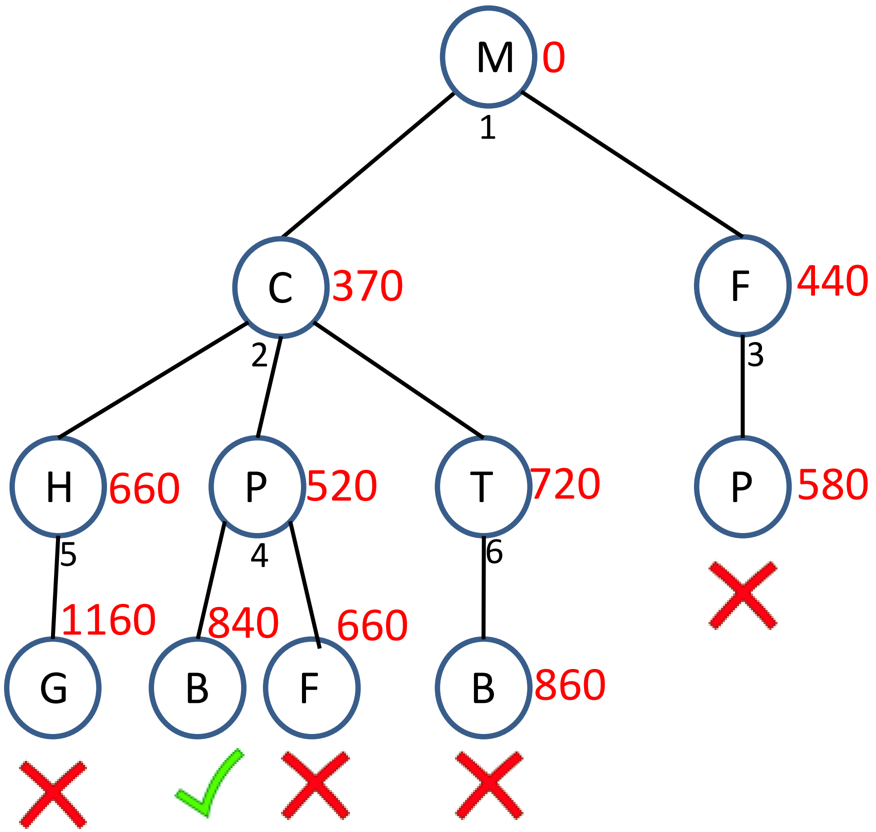


- Deals with search situations in which the cost of traversing a path is of a primary importance
- One procedure -> find all possible paths (depth first or breadth first) and select the best one -> works good for small search trees
- If the search tree is large we must apply an heuristic search
- One solution – branch and bound

Branch and bound search – The algorithm

- Initialize the queue with the *Initial state*, having a zero-length path
- **REPEAT** until at least one path in the queue terminates at the goal node OR queue is empty
    - Remove the first path from queue -> *firstPath*
    - Extends the *firstPath* to all the neighbours (except the neighbors that are already in the *firstPath*) of the terminal node -> *newPaths*
    - Add the *newPath* (if exists) to the queue
    - Sort the entire queue by path length with least-cost paths in front
- If the goal is found return SUCCESS

Branch and bound + Extended List

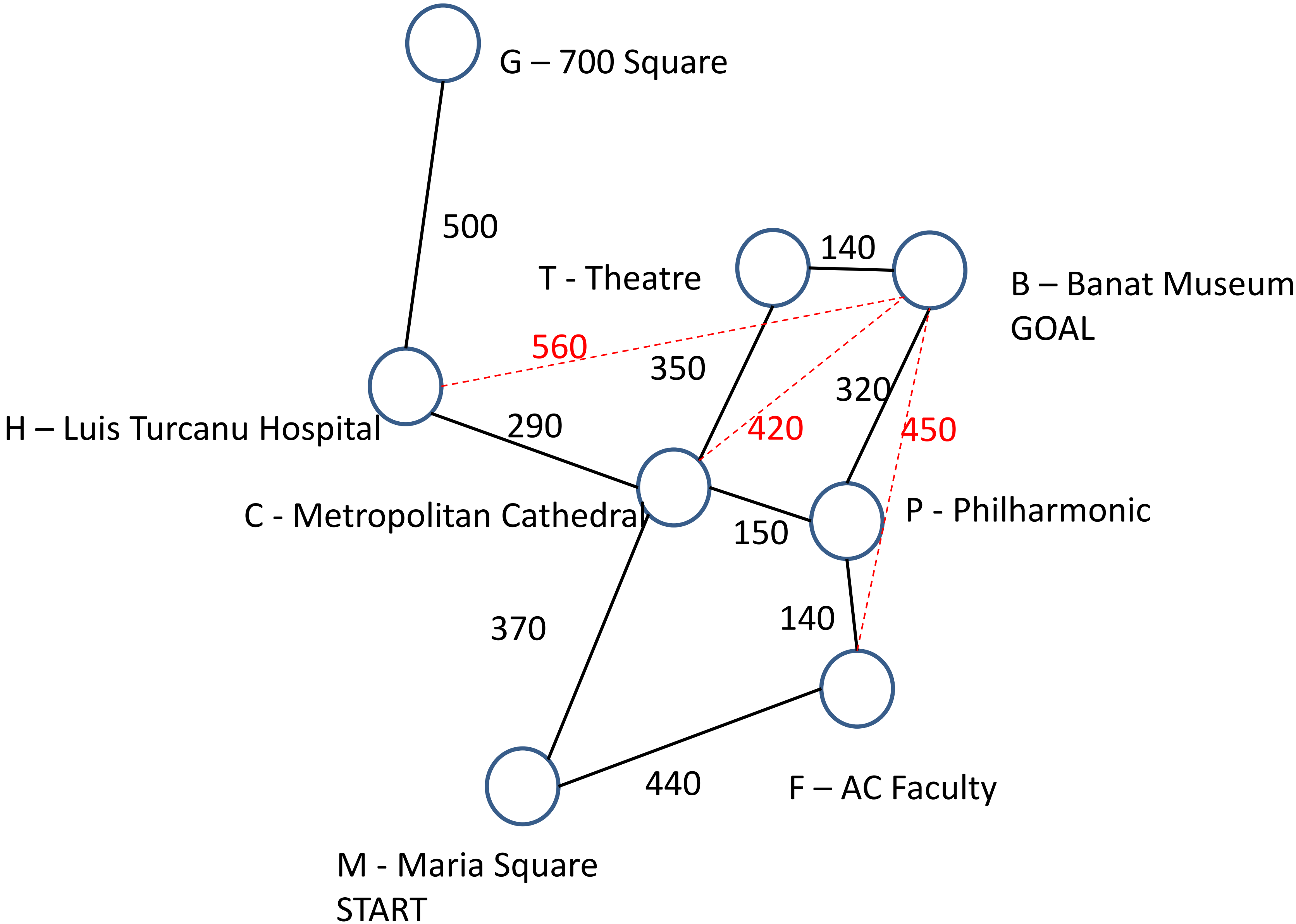


M C F P H T

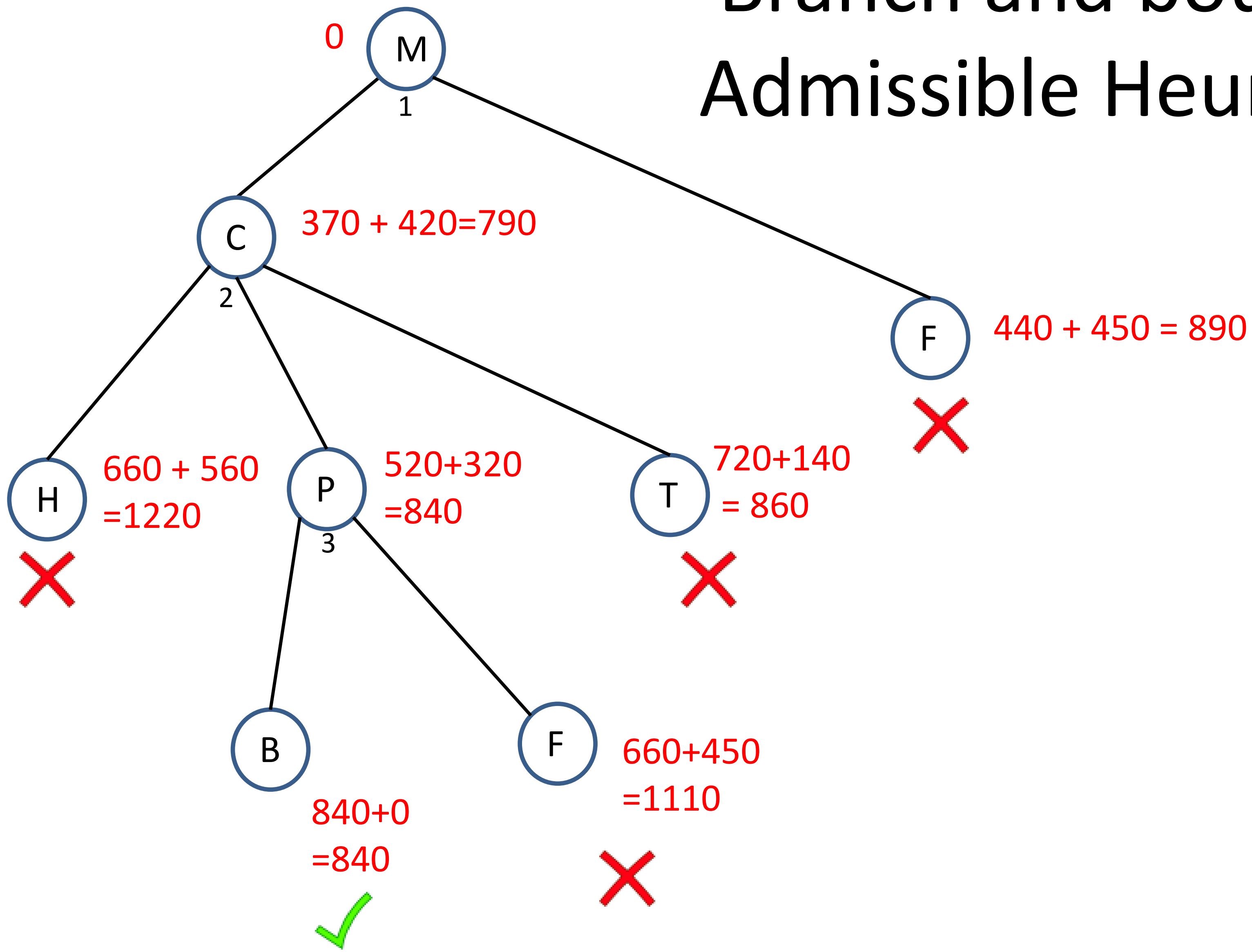
Branch and bound + Admissible heuristics

- We can improve the bound and branch search using guesses about the remaining distance from a node to the goal
- If our guess is good we have an estimation (*e*) of the total path length:
  - *e(total path length) = d(already traversed) + e(distance remaining)*
- In most of the cases guesses are not perfect, and a bad overestimate can cause errors (the right path can be overlooked)
- Admissible heuristics – heuristics that underestimate(*u*) the path; it's always smaller or equal than the real distance
  - *u(total path length) = d(already traversed) + u(distance remaining)*
- When the algorithm found a total path, you don't need to go further because all partial-path distance estimates are longer that the total path you already found (we *underestimate* the distances)

Problem – Navigation in Timisoara



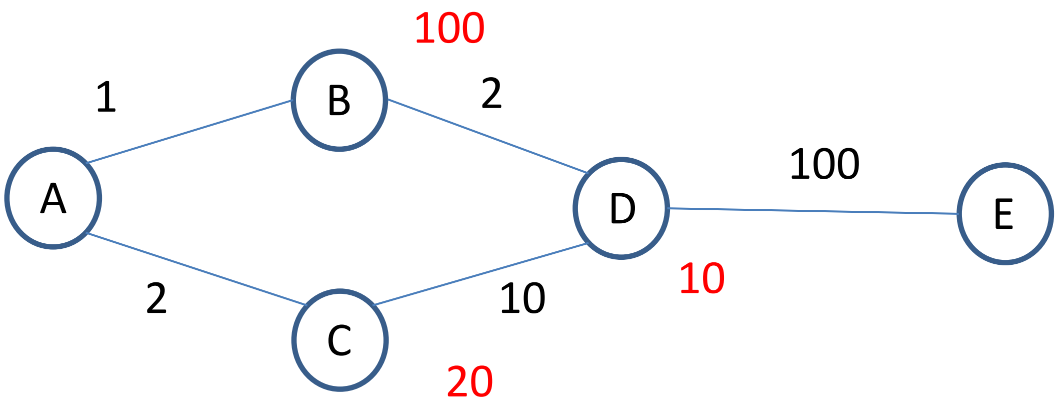
Branch and bound + Admissible Heuristics

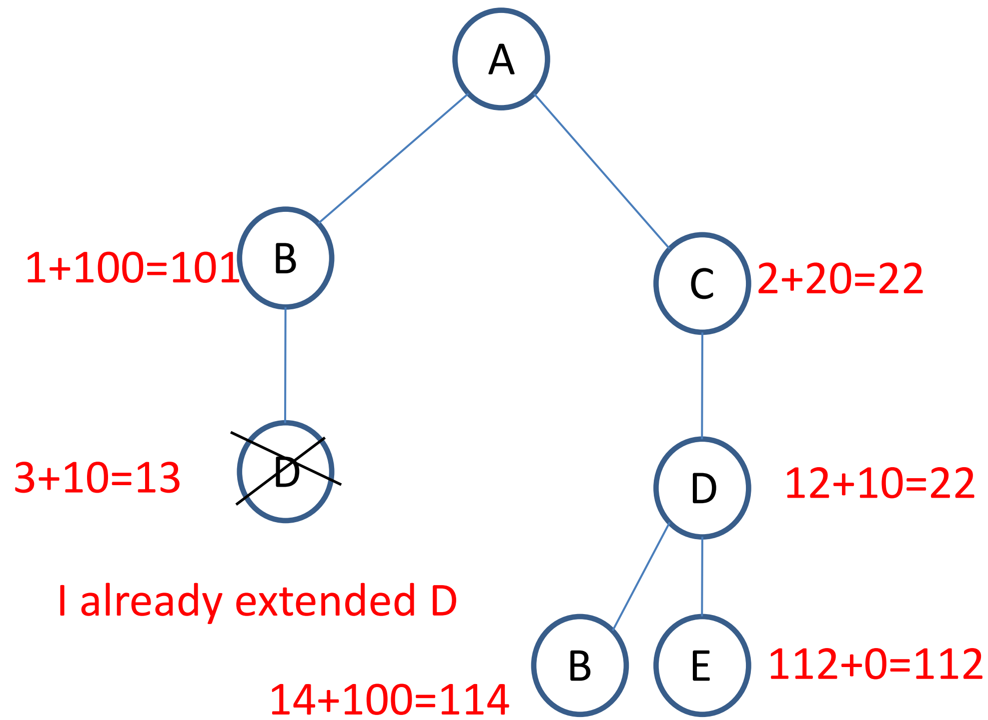


A*

- A* = Branch and bound + Extended list + Admissible heuristics
- Initialize the queue with the *Initial state*, having a zero-length path
- **REPEAT** until at least one path in the queue terminates at the goal node OR queue is empty
    - Remove the first path from queue -> *firstPath*
    - Extends the *firstPath* to all the neighbours (except the neighbors that are already in the *firstPath*) of the terminal node -> *newPaths*
    - Add the *newPath* (if exists) to the queue
    - Sort the entire queue by the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost in front
- If the goal is found return SUCCESS

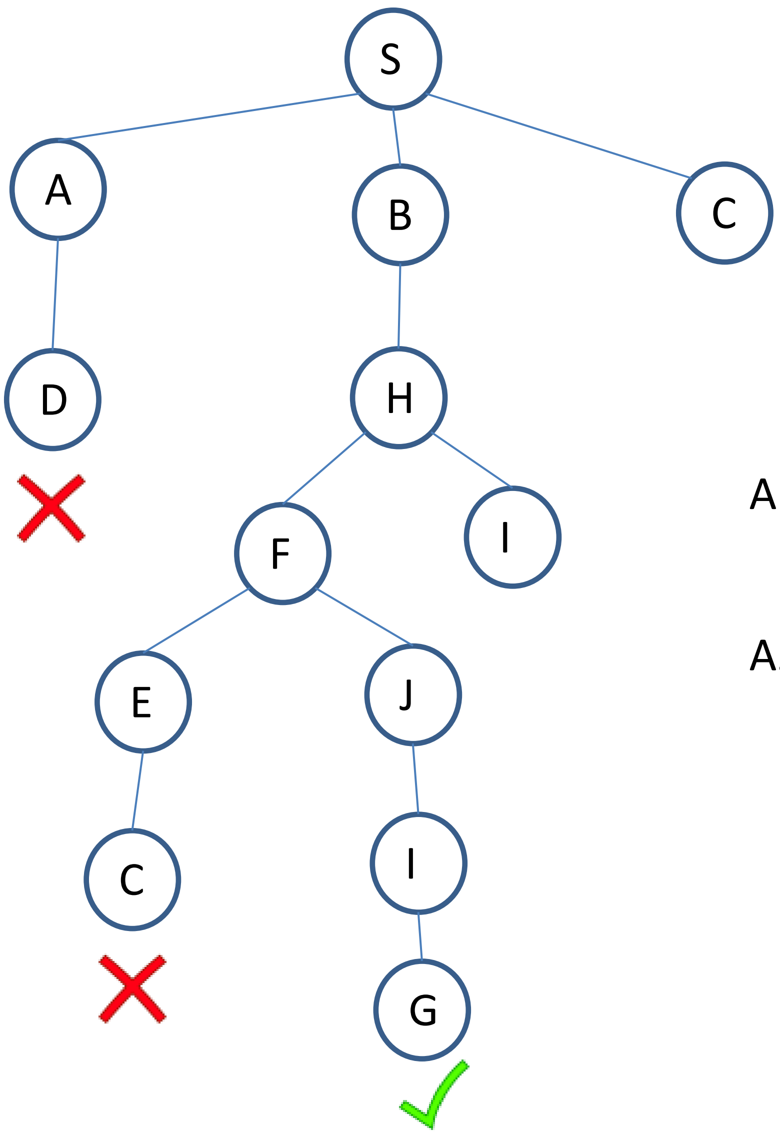
A* - admissibility problem





- ADMISSIBLE heuristic if the estimated distance (H) between any node X and the goal G is less or equal to the actual distance between the X and the goal (works if it’s a map)
    - $H(X,G) \leq D(X,G)$
- CONSISTENCY for every 2 nodes X and Y if
    - $| H(X,G) – H(Y,G) | \leq D(X,Y)$

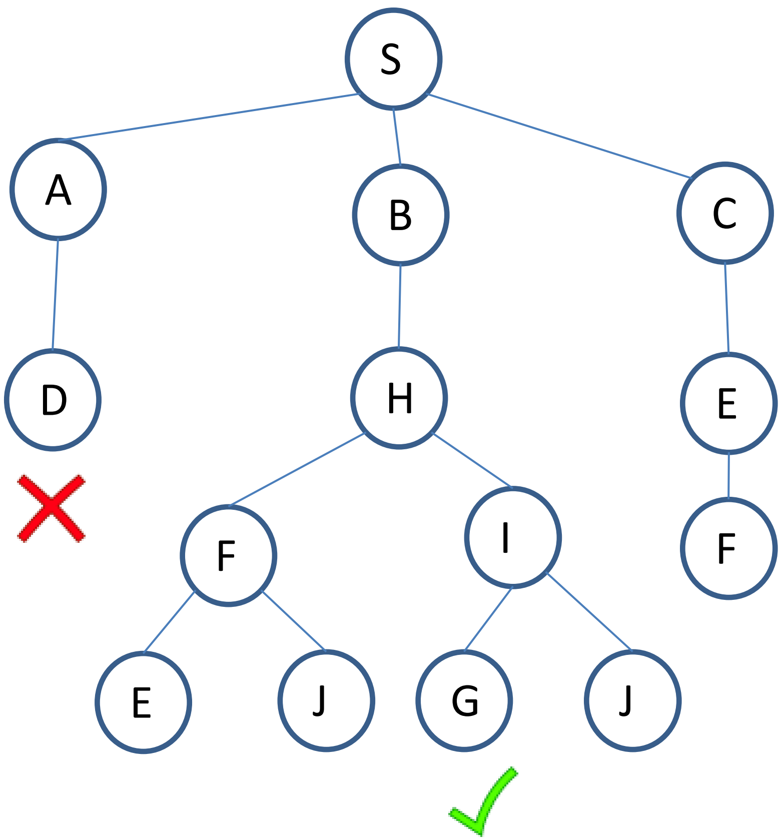
Sample search exam problem



A1 – S , A , D , B , H , F , E , C , J , I

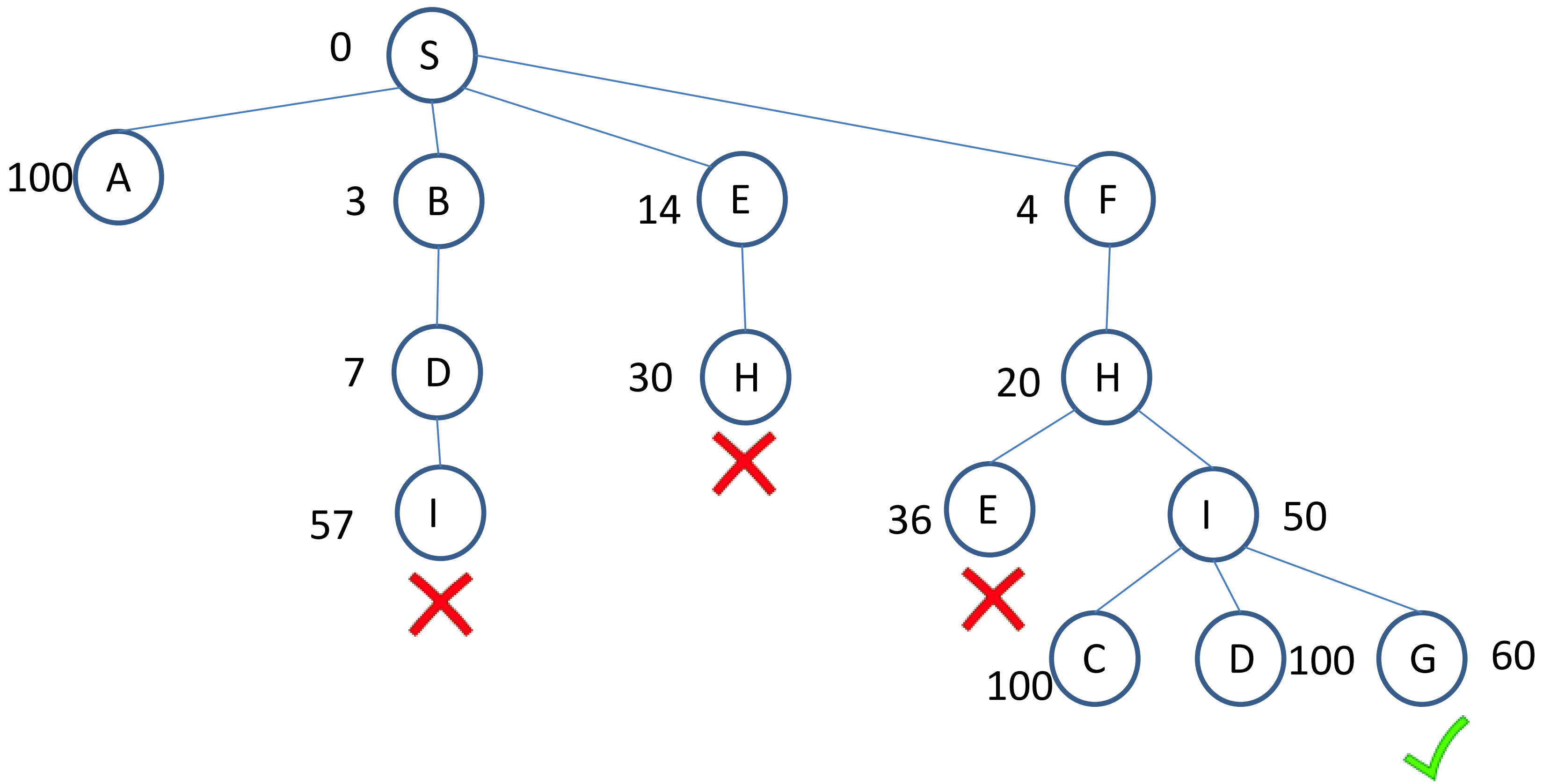
A2 – 2 times

Sample search exam problem



A3 – S , A , B , C , D , H , E , F , I

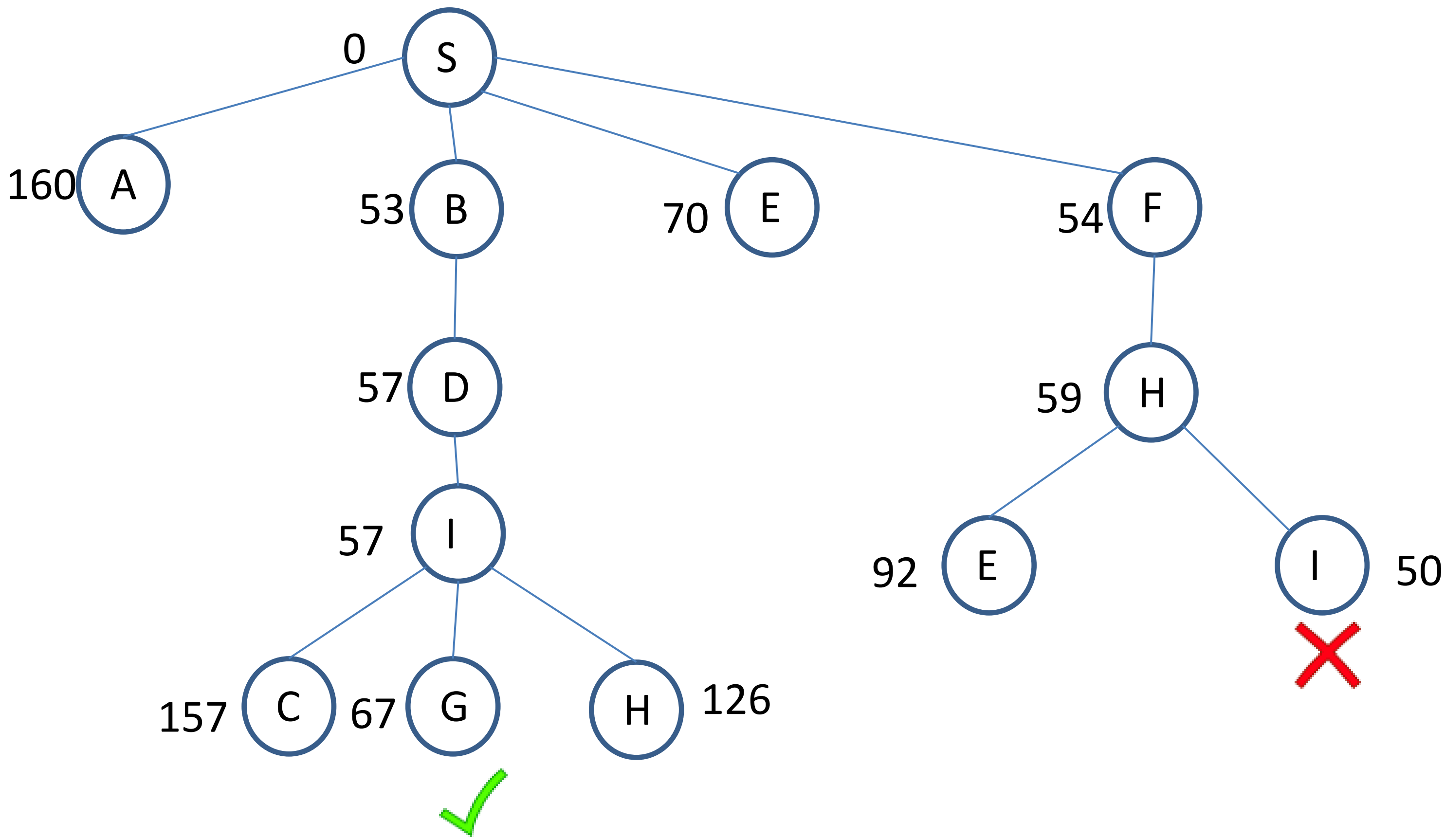
Sample search exam problem



B1 – S , B , F , D , E , H , I

S – F – H – I - G

Sample search exam problem



B2 – S , B , F , D , I , H
S – B – D – I - G

B3 - Heuristics is not consistent.

Related resources

- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/exams/MIT6_034F10_quiz1_2008.pdf

Readings

- Artificial Intelligence (3rd Edition), Patrick Winston, Chapter 4 and Chapter 5