Dealing with Exponential Problems

- Many important problems have no polynomial solution but we still need to deal with them.
- The approach is to get "good" solutions in "efficient" time.
- Goodness and efficiency can be traded off to a certain degree.
- A number of powerful approaches:
  - Intelligent Search
  - Approximation Algorithms
  - Local Search
  - Stochastic Algorithms
- The term "approximation algorithms" is often used for many of these approaches trying to approximate the optimal solution. Some intelligent search techniques will guarantee the optimal solution, but the search is still exponential (as you would expect).
Intelligent Search

The full problem is made up of an exponential search space and thus exhaustive search is exponential.

Informed (intelligent) vs. uninformed search.

Intelligent search seeks to reduce the number of states we actually search by one or both of the following:

- Pruning, as soon as possible, branches of the search tree which cannot lead to a solution.
- Trying first those branches which are most promising in hopes of finding a reasonable solution quickly, thus avoiding alternatives.

Approaches:
- Backtracking
- Branch and Bound
- Beam Search – Non optimal solution, but faster
- A* search
Backtracking

- Natural for decision search problems
  - Is there a solution or not

- SAT – Satisfiability
  - Does there exist a setting of Boolean variables which satisfies a particular CNF Boolean expression
  - $2^n$ possible settings
  - NP-complete problem
  - Verifiable in polynomial time – Easy to test
    - What is simple algorithm/complexity?
  - What does full search tree look like?
  - Backtracking allows us to prune branches
  - Branch on arbitrary variable (e.g. $w$)
  - Both leafs still possible, but when we branch on $x$, note that the leaf ($w=0, x=1$) is impossible so we can prune all subtrees of that node – can backtrack out of that dead-end
• Clauses in a node represent what still remains to be satisfied
• Dropping a clause in a node means that the clause is satisfied
• If we find any empty node (i.e. all clauses satisfied) then full expression is satisfiable
• An empty clause signifies that the clause is not satisfiable and we can prune that path
• If all leafs contain an empty clause then the expression is not satisfiable
Backtracking Algorithm

Start with some problem $P_0$
Let $S = \{P_0\}$, the set of active subproblems
Repeat while $S$ is nonempty:
  - choose a subproblem $P \in S$ and remove it from $S$
  - expand it into smaller subproblems $P_1, P_2, ..., P_k$
  For each $P_i$:
    - If $\text{test}(P_i)$ succeeds: halt and announce this solution
    - If $\text{test}(P_i)$ fails: discard $P_i$
    - Otherwise: add $P_i$ to $S$
  Announce that there is no solution

- Backtracking is still exponential, but due to the pruning it is more practical in many cases
  - More room to increase the size of the problem before it becomes intractable
Branch and Bound

- Similar to Backtracking but applied to optimization problems
- Common and powerful technique for many tasks
- For a minimization problem, we need an efficient/quick mechanism to generate an optimistic lowerbound $\mathcal{L}(S)$ for a particular state $S$ (or upperbound for maximization)
  - From $S$ we know that we could never get a better solution than $\mathcal{L}(S)$ – Note we want $\mathcal{L}(S)$ as big as possible
- We keep track of the Best Solution So Far ($BSSF$)
- If $\mathcal{L}(S) \geq BSSF$ then we can prune $S$ and not search anymore below it
- The better (tighter) the bounding function the more we can prune the search
Branch and Bound Algorithm

Start with some problem $P_0$
Let $S = \{P_0\}$, the set of active subproblems
bestsofar = $\infty$

Repeat while $S$ is nonempty:
- choose a subproblem (partial solution) $P \in S$ and remove it from $S$
- expand it into smaller subproblems $P_1, P_2, ..., P_k$
For each $P_i$:
  - If $P_i$ is a complete solution: update bestsofar
  - else if $\text{lowerbound}(P_i) < \text{bestsofar}$: add $P_i$ to $S$
return bestsofar

- B&B returns the optimal solution
- Just like backtracking, B&B is still exponential, but often runs in relatively fast time due to pruning and depending on the quality of the bounding function
- Before expanding a problem $P \in S$, make sure its' LB is still $< BSSF$, since $BSSF$ may have been updated since $P$ was put in the set $S$
- Also, as a heuristic, could make $S$ a priority queue with $\text{LB}(P)$ as the key – always expand $P$ with lowest LB as a heuristic
  - All $P$ with $\text{LB} < BSSF$ must be expanded before we terminate, to be optimal
Want the least number of moves necessary to complete the puzzle

What does the search space look like?

What kind of bounding function could we use to help prune space?

- Must never over-estimate the actual optimum (is a lower-bound)
- Can we do better?
- The closer the bound can get to the actual optimum, the more we can prune
- The bounding function should be relatively fast to compute
- Would if bound estimate overestimated needed steps from the state?
A beam search is a semi-greedy state space search that only keeps a limited specified number (the beam width) of states in memory.

When the state space exceeds the beam width, then all the states with the worst bound/heuristic value are dropped.

Thus it is sub-optimal since the optimal solution may have been in one of the dropped states.

More efficient in both time and space:
- Memory is controlled explicitly by the beam width.

Can make B&B a beam search by putting a max (beam width) on the number of states to save. Common approach.
The complete solutions are the \((n-1)!\) leaves
- One for each possible path

Would like an efficient bounding function for each state which could allow pruning to avoid searching all states
- What is the loosest and what is the tightest bound?
- What are some other bounds?
Would like an efficient bounding function for each state which could allow pruning to avoid searching all states
- Current partial distance + sum of the shortest edges leaving remaining nodes
- Current partial distance + MST(remaining nodes)
- etc.

Note that we also need a search space strategy that drills down to some final states so we can keep updating $BSSF$ to allow better pruning
Reduced Cost Matrix Approach

- As long as the bound is optimistic (≤ the optimal tour), branch and bound will return the optimal path for TSP.
- You will implement this bounding approach for your project, together with an improved state space search.
- We will not give you exact pseudo-code for this project, but will review approaches in the slides and make available some pertinent reading material.
- $n$ city problem – Directed graph version – can have asymmetric city distances.
- TSP is a Hamiltonian Cycle, also called a Rudrata Cycle.
Hint: Every tour must leave every vertex and arrive at every vertex. Any proposals?
Bound on TSP Tour

- Could use sum of smallest exiting edge from each vertex
- or similarly, Could use sum of smallest entering edge into each vertex
  - Are these optimistic?
- Can do even better if we can combine both
  - Sum them?
  - Do one first, then somehow sum in the residual of the other, given the first
What’s the cheapest way to leave each vertex?
Save the sum of those costs in the bound (as a first draft). Still can gain better bound by making sure we arrive at each vertex.

Initial rough bound $= 8 + 2 + 3 + 6 + 1 = 20$
For a given vertex, subtract the least departure edge cost from each edge leaving that vertex. This leaves the potential increased cost of taking a different edge in the reduced cost matrix. If we decide later to use edge (1,2) instead of (1,4), the cost would be only 1 more (9) than we already had (8).
Repeat for the other vertices. Now, let's find a tighter lower bound. Does the set of edges now having 0 residual cost (those making up our current bound) arrive at every vertex? – No. None into vertex 3.
We have to take an edge to vertex 3 from somewhere. Assume we take the cheapest currently available.
Subtract this cheapest edge cost from all edges entering vertex 3 and add the cost to the bound. We do this for all vertices which did not have an in-bound edge of 0 cost to complete our reduced cost matrix. We have just tightened the bound.
The Bound

- It will cost at least this much to visit all the vertices in the graph.
  - There is no cheaper way to get in and out of each vertex.
  - The edges in our reduced cost matrix are now labeled with the extra cost of choosing that edge.
- Remember, the bound is not necessarily (and not usually) a solution, and for minimization is always ≤ optimal solution.
We can accomplish the steps of this algorithm using a cost matrix.
Reduced Cost Matrix

Reduce all rows by subtracting the minimum path from the other path lengths. Sum the reduction amounts as we go to build the bound.

Initial rough bound
= 8 + 2 + 3 + 6 + 1 = 20
Reduced Cost Matrix

Then reduce all columns by subtracting the minimum path lengths and adding them to the bound. Only column 3 reduces since it has no 0's. Now we have a tighter bound. Note that there is now at least one 0 in every row and column.

final bound = 20 + 1 = 21
Reduced Cost Matrix Algorithm

bound = 0 initially (or is the bound of the parent state)

For each row in $A$
  Subtract the minimum cell value in the row from all cells in the row
  and add that value to the bound

Then for each column in $A$
  Subtract the minimum cell value in the column from all cells in the column
  and add that value to the bound

At that point every column and row will have at least one 0 entry
and we will have the correct bound summed

\[
\begin{bmatrix}
\infty & 20 & 30 & 10 & 11 \\
15 & \infty & 16 & 4 & 2 \\
3 & 5 & \infty & 2 & 4 \\
19 & 6 & 18 & \infty & 3 \\
16 & 4 & 7 & 16 & \infty \\
\end{bmatrix}
\]

(a) Cost matrix

\[
\begin{bmatrix}
\infty & 10 & 17 & 0 & 1 \\
12 & \infty & 11 & 2 & 0 \\
0 & 3 & \infty & 0 & 2 \\
15 & 3 & 12 & \infty & 0 \\
11 & 0 & 0 & 12 & \infty \\
\end{bmatrix}
\]

(b) Reduced cost matrix

$L = 25$
Reduced Cost Matrix Algorithm

- Could have done columns first and then rows
  - Which is better?
- Still give a correct lower bound, but not necessarily the same reduced cost matrix as the row first approach
  - Could try both, but then we have the trade-off of more time vs finding a better bound
Initial BSSF

Initial BSSF Value: Infinite?

A tighter value would lead to more initial pruning. Should be quick to calculate, but best if it is a reasonable value, since it may take a while before your first complete solutions finish and allow BSSF update.

Don't use reduced matrix value because it will be lower than the optimal and algorithm would not be complete. BSSF is opposite of the bound in that it must be $\geq$ the optimal path (upper bound) - Want BSSF as small as possible.

Could try a few random legal paths and pick the best. Be creative on this for your project. Can make a significant difference.
Random \( BSSF \)

Cost of random \( BSSF \)
\[
= 9 + 5 + 4 + 12 + 1 = 31
\]

Want \( BSSF \) as low as possible and \( LB \) as high as possible

Where does optimal lie?
The state space search approach defines the manner in which children (or “successor”) states are expanded from a given state in the state space search (i.e. How we search through the tree). This is the search "frontier," which in this case is stored in the priority queue.

- We will introduce two approaches

The first assumes each state represents a partial path
- A link to a child state represents a path from the root city to the child
- We will arbitrarily make the first state represent city 1

Children states in the search tree are generated by considering each path leaving the parent city in the TSP graph. We then calculate the reduced cost matrix for each child state and put child states whose bound is less than the BSSF on a priority queue where the bound is the PQ key
- Do we need to use a priority queue?

Note that the initial reduced cost matrix is the same regardless of which node we started at.
Partial Path State Space

Figure 8.10 State space tree for the traveling salesperson problem with \( n = 4 \) and \( i_0 = i_4 = 1 \)
State Space Search – Partial Path

State 1 (21)

Don't confuse search state number with city number

What changes?

bound = 21 + ?
State Space Search – Partial Path

Don't confuse search state number with city number

What changes? (row:from column:to)
bound = 21 + 1 (cost of edge) + ?
First set "from row" and "to column" = ∞
also for selected edge \((i,j)\), set \((j,i) = ∞\)
State Space Search – Partial Path

State 1 (21)

\[ \begin{array}{cccc}
\infty & 1 & \infty & 0 \\
\infty & \infty & 1 & \infty \\
\infty & 0 & \infty & 1 \\
0 & \infty & \infty & 9 \\
\end{array} \]

\[ BSSF = 31 \]

Don't confuse search state number with city number

What changes? (row:from column:to)

\[ bound = 21 + 1 + 1 = 23 \]

\[ = \text{prev bound} + \text{path}(1,2) + \text{cost of reducing the updated cost matrix} \]
State Space Search – Partial Path

State 1 (21)

\[
\begin{array}{cccc}
\infty & 1 & \infty & 0 & \infty \\
\infty & \infty & 1 & \infty & 0 \\
\infty & 0 & \infty & 1 & \infty \\
\infty & 0 & 0 & \infty & 6 \\
0 & \infty & \infty & 9 & \infty \\
\end{array}
\]

\(BSSF = 31\)

Don't confuse search state number with city number

State 2 (21+1+1=23)

State 3 (21+\(\infty\)+1=\(\infty\))

State 4 (21+0+0=21)

State 5 (21+\(\infty\)+\(\infty\)=\(\infty\))
State Space Search

- To create each new state, start with the bound for the parent state and
  - Add the residual cost of the new path
  - Set to infinity paths which can no longer be used (row of from-state, column of to-state, and to/from state)
  - Reduce the new matrix and add the cost of reduction

- States 3 and 5 will be pruned
- States 2 and 4 will be enqueued and state 4 (with the lowest bound 21) will be the first on the queue and the next state expanded if using a priority queue
  - Adjust BSSF? - Not until we have a complete solution which is better
- Number of remaining edges in the graph decrease at each level of the tree
State Space Search

State 4 (21)

\[
\begin{array}{cccccc}
\infty & \infty & 1 & \infty & 0 \\
\infty & 0 & \infty & \infty & \infty \\
\infty & 0 & 0 & \infty & 6 \\
0 & \infty & \infty & \infty & \infty \\
\end{array}
\]

Went from city 1 to 4

\[BSSF = 31\]
State Space Search

State 4 (21)

\[
\begin{array}{cccccc}
\infty & \infty & 1 & \infty & 0 \\
\infty & 0 & \infty & \infty & \infty \\
\infty & 0 & 0 & \infty & 6 \\
0 & \infty & \infty & \infty & \infty \\
\end{array}
\]

\(BSSF = 31\)

State 12 (21+0+\infty=\infty)

State 13 (21+0+0=21)

State 14 (21+6+1)=28

13 and 14 will be put on the queue with State 13 the lowest
State Space Search

State 13 (21) Path so far 1, 4, 3

\[
\begin{pmatrix}
\infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & 0 & \infty \\
\infty & 0 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty \\
0 & \infty & \infty & \infty & \infty \\
\end{pmatrix}
\]

Just the first state is put in the queue and it will then select path 2-5 with no new added cost. The last step will be the path 5-1 which also has no added cost.

\[BSSF = 31\]
Complete path at bottom state is
1 – 4 – 3 – 2 – 5 – 1  Total Distance = 21

Once we hit the first complete solution, BSSF is updated to 21

In this case the two states on the queue (23, 28) will be pruned as soon as they are dequeued, leaving the queue empty and the algorithm will stop and return 21

If there had been states on the queue with a bound lower than the updated BSSF, then processing would have continued
Partial Path B&B Complexity

- Number of node expansions required?
Partial Path B&B Complexity

- Number of node expansions required? $O(b^n)$ where $b$ is the average state branching factor and $n$ is the depth of solutions
  - Note it is not a balanced tree, but the frontier still grows exponentially
  - The hope is that pruning will keep the branching factor down and also keep the tree more shallow in many places

- Space Complexity?
Partial Path B&B Complexity

- Number of node expansions required? $O(b^n)$ where $b$ is the average state branching factor and $n$ is the depth of solutions
  - Note it is not a balanced tree, but the frontier still grows exponentially
  - The hope is that pruning will keep the branching factor down and also keep the tree more shallow in many places

- Space Complexity? $O(n^2b^n)$ since we store one reduced cost matrix ($n^2$) for each node in the queue and the queue is the frontier of the search tree which is $O(b^n)$

- Time Complexity?
Partial Path B&B Complexity

- Number of node expansions required? $O(b^n)$ where $b$ is the average state branching factor and $n$ is the depth of solutions
  - Note it is not a balanced tree, but the frontier still grows exponentially
  - The hope is that pruning will keep the branching factor down and also keep the tree more shallow in many places

- Space Complexity? $O(n^2b^n)$ since we store one reduced cost matrix ($n^2$) for each node in the queue and the queue is the frontier of the search tree which is $O(b^n)$

- Time Complexity? – $O(n^2b^n)$, the number of nodes created for potential queue insertion, times the complexity of creating a node which is the cost of reducing the cost matrix which is $O(n^2)$
Improved State Space Search

- We used a priority queue so that the state with the best (lowest) bound can be chosen when it is time to expand a new state
  - For the partial path state space search this can lead to a fairly broad (breadth-first) search frontier
  - All $O(n)$ child states are calculated (and many put in the queue) each time a state is expanded
  - Leads to a large number of states in the priority queue
  - Also, $BSSF$ is not updated until a completed path node (leaf) is reached which could be a while for large $n$

- Can we use a different state space to gain advantage?
  - We can still use the same bounding scheme – reduced cost matrix
  - Want a state space search that "drills deeper" leading to finding complete paths which will update the $BSSF$ sooner and lead to earlier pruning
Include/Exclude State Space Search

- Include/Exclude approach improves TSP performance compared to the partial path approach and you will use it for your project.
- We'll use the same reduced cost matrix bounding function, though any bounding function could be used.
- We do not choose an initial city to start at; we just have the initial reduced cost matrix tied to the initial search state.
- The search tree will be a binary tree with the decision based on whether a particular edge is in the final solution or out of it.

\[
\text{Include edge } (i,j) \quad \quad \quad \quad \quad \text{Exclude edge } (i,j)
\]

- Note that the left (include) child reduces our subsequent options to paths of length \( n-1 \) while the right (exclude) child still requires us to find a path of length \( n \), which cannot include edge \((i,j)\).
- We want to include the best edges (usually very short edges between cities). Solutions including the good edges usually are better and have low lower bounds. Solutions which exclude that good edge are usually worse and have higher lower bounds. \( b(\text{exclude}) - b(\text{include}) \) can find the edge which maximizes that difference.
- We hope this leads to "digging deep" down the include side to get new BSSFs, while leading to quick pruning down the exclude side.
Include/Exclude State Space Search

How do we select one of the up to $n^2$ edges in our graph?

- Select from those in our current reduced cost matrix – still $n^2$
- Including 0 cost paths tends to keep include bounds low, while excluding 0 cost paths tends to increase exclude bounds, (and always at least one 0 cost path available in every row not yet travelled from), thus we will only consider 0 cost paths – closer to $O(n)$ edges to consider

Select the edge which maximizes $b(S_e) - b(S_i)$ (b is bound function)

- Assume that we chose edge (5,1) – brown fill represents included path, green represents updates

$S_i(21)$

$S_e(21 + (9+∞)) = ∞$

$S_1(21)$
Which Edge is Best?

- How do we find the best edge to choose?
- The candidate (0 cost) edges are (1,4), (2,5), (3,2), (4,2), (4,3), (5,1)
- We can just try them all ($O(n)$) to see which maximizes $b(S_e) - b(S_i)$
- Note that after setting $S_e(i,j) = \infty$ that
  \[
  b(S_e) = b(S_{parent}) + \min(\text{row}_i) + \min(\text{column}_j)
  \]

We reduce both states $S_e$ and $S_i$ and put them in the priority queue (as long as their bound < $BSSF$) and then take next lowest state off the queue $S_i$ and children of $S_i$ must keep track that (5,1) is part of its final path

Note that we can free or reuse memory for $S_1$ at this point

---

Include edge (5,1)

$S_i(21+0+0 = 21)$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$\infty$</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Exclude edge (5,1)

$S_e(21+0+(9+\infty)) = \infty$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$\infty$</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

$S_1(21)$
Premature Cycles

- One last item which must be dealt with: Premature cycles
- If edge \((i,j)\) is included, then edge \((j,i)\) can not be used and must be marked as infinite in \(S_i\)
  - In the case below \((5,1)\) it already happened to be infinite, but normally may not be
- In fact, we must set to infinity all edges \((k,m)\) which would complete a consecutive partial solution (length < \(n\)) starting from \(j\) and ending at \(i\). In the picture below assume \(n > 5\) and that each white arrow is an included edge along an include state-space path and we then add the brown edge. We must set to infinity all the red edges shown.
- These exclusions may cause further reduction of \(S_i\) which must be considered when calculating the reduced cost matrix for \(S_i\)

\[
\begin{array}{cccc}
\infty & 1 & \infty & 0 \\
\infty & \infty & 1 & 0 \\
0 & \infty & 1 & \infty \\
0 & 0 & \infty & 6 \\
0 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty \\
\end{array}
\]

\[
\begin{array}{cccc}
\infty & \infty & 1 & 0 \\
\infty & \infty & 1 & \infty \\
0 & \infty & 1 & \infty \\
0 & 0 & \infty & 6 \\
0 & \infty & \infty & 9 \\
\infty & \infty & \infty & \infty \\
\end{array}
\]

\[
\begin{array}{cccc}
\infty & \infty & 0 & \infty \\
\infty & \infty & 0 & \infty \\
\infty & \infty & 1 & \infty \\
0 & \infty & \infty & 6 \\
\infty & \infty & \infty & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
\infty & 0 & \infty & 6 \\
\infty & \infty & \infty & 0 \\
\infty & 0 & \infty & 6 \\
\infty & \infty & \infty & 0 \\
\end{array}
\]
Premature Cycles

- One last item which must be dealt with: Premature cycles
- If edge \((i, j)\) is included, then edge \((j, i)\) can not be used and must be marked as infinite in \(S_i\)
  - In the case below \((5,1)\) it already happened to be infinite, but normally may not be
- In fact, we must set to infinity all edges \((k,m)\) which would complete a consecutive partial solution (length < \(n\)) starting from \(j\) and ending at \(i\). In the picture below assume \(n>5\) and that each white arrow is an included edge along an include state-space path and we then add the brown edge. We must set to infinity all the red edges shown.
- These exclusions may cause further reduction of \(S_i\) which must be considered when calculating the reduced cost matrix for \(S_i\)

![Diagram](image-url)
One last item which must be dealt with: Premature cycles

If edge \((i, j)\) is included, then edge \((j, i)\) cannot be used and must be marked as infinite in \(S_i\)

In the case below \((5,1)\) it already happened to be infinite, but normally may not be

In fact, we must set to infinity all edges \((k, m)\) which would complete a consecutive partial solution (length < \(n\)) starting from \(j\) and ending at \(i\). In the picture below assume \(n>5\) and that each white arrow is an included edge along an include state-space path and we then add the brown edge. We must set to infinity all the red edges shown.

These exclusions may cause further reduction of \(S_i\) which must be considered when calculating the reduced cost matrix for \(S_i\)

\[
\begin{array}{cccc}
\infty & 1 & \infty & 0 \\
\infty & \infty & 1 & \infty \\
\infty & 0 & \infty & 1 \\
0 & 0 & \infty & 6 \\
0 & 0 & \infty & \infty \\
\infty & \infty & \infty & \infty
\end{array}
\]
One last item which must be dealt with: Premature cycles

If edge \((i,j)\) is included, then edge \((j,i)\) can not be used and must be marked as infinite in \(S_i\)

In the case below \((5,1)\) it already happened to be infinite, but normally may not be

In fact, we must set to infinity all edges \((k,m)\) which would complete a consecutive partial solution (length < \(n\)) starting from \(j\) and ending at \(i\). In the picture below assume \(n>5\) and that each white arrow is an included edge along an include state-space path and we then add the brown edge. We must set to infinity all the red edges shown.

These exclusions may cause further reduction of \(S_i\) which must be considered when calculating the reduced cost matrix for \(S_i\)
Avoiding Premature Cycles

If you add edge \((i,j)\) then you need to set to infinity (delete) some edges that are subsequently impossible and might lead to a premature cycle. The vectors Entered and Exited are initialized to -1 before processing is started and updated as processing continues. This should do the job, since other premature cycles are prevented by our matrix updates, but you may do it however you want.

function deleteEdges (M: matrix, i,j: int): matrix
    Entered[j] = i
    Exited[i] = j
    start = i
    end = j
    // The new edge may be part of a partial solution. Go to the end of that solution.
    while (Exited[end] != -1) do end = exited[end]
    // Similarly, go to the start of the new partial solution.
    while (entered[start] != -1) do  start = entered[start]
    // Delete the edges that would make partial cycles, unless we’re ready to finish the tour
    if (partial_path_length < n-1) then
        Repeat Until (start = j)
            M[end, start] = infinity
            M[j, start] = infinity
            start = exited[start]
    return M
Premature Cycles in Partial Path

- For the Partial Path State Space Search you do not need to worry about premature cycles, except for setting edge \((j,i)\) to infinity when we are adding edge \((i,j)\)
  - Even that is only useful at the initial expansion of the root node.

- This is because any attempt to expand further will not happen, because the entire row \(i\) will have been set to infinity when \((i,j)\) was initially expanded.

- Also, if desired, you could just have a test in the code which would not open any child which is already on the path in that branch of the tree
  - Unless it is the last child (which must be node 1) and the path will now be length \(n\) (a complete solution)
One More Iteration

- Possible zero edges are?

<table>
<thead>
<tr>
<th>Include edge (5,1)</th>
<th>Exclude edge (5,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_i(21+0+0 = 21)$</td>
<td>$S_e(21+0+(9+\infty)) = \infty$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>9</th>
<th>\infty</th>
<th>\infty</th>
<th>\infty</th>
<th>\infty</th>
<th>\infty</th>
<th>\infty</th>
<th>\infty</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>6</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>6</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
<td>0</td>
<td>$\infty$</td>
<td>6</td>
<td>0</td>
<td>9</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Include edge (5,1)
$S_i(21+0+0 = 21)$

Exclude edge (5,1)
$S_e(21+0+(9+\infty)) = \infty$
One More Iteration

- Possible zero edges are (1,4), (2,5), (3,2), (4,2), (4,3)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th></th>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td></td>
<td>∞</td>
<td></td>
<td></td>
<td>∞</td>
</tr>
<tr>
<td>∞</td>
<td></td>
<td>1</td>
<td>∞</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td></td>
<td>1</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>∞</td>
</tr>
</tbody>
</table>

Include edge (5,1)

$S_i(21+0+0 = 21)$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th></th>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td></td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td></td>
<td>1</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>9</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

Exclude edge (5,1)

$S_e(21+0+(9+∞)) = ∞$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th></th>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td></td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td></td>
<td>1</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>1</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_1(21)$
One More Iteration

- Possible zero edges are (1,4), (2,5), (3,2), (4,2), (4,3)
- Choosing (2,5) maximizes $b(S_e) - b(S_i)$
- To avoid premature cycles in $S_i$ we set (5,2) (no change in this case) and (1,2) to $\infty$ - Why? note (1,5) was set $\infty$ in the previous iteration
- Both states are added to the queue assuming BSSF is 31 as before

Include edge (2,5)

$S_i (21+0+0=21)$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>$\infty$</th>
<th>0</th>
<th>$\infty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>0</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\infty$</td>
<td>6</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

$S_2 (21)$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>$\infty$</th>
<th>0</th>
<th>$\infty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
<td>0</td>
<td>$\infty$</td>
<td>6</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Exclude edge (2,5)

$S_e (21+0+(6+1)=28)$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>$\infty$</th>
<th>0</th>
<th>$\infty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
<td>0</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

Now includes edges (5,1) and (2,5)
Include/Exclude Search

Start with some problem $P_0$
Let $S = \{P_0\}$, the set of active subproblems
bestsofar = $\infty$
Repeat while $S$ is nonempty:
  choose a subproblem (partial solution) $P \in S$ and remove it from $S$
  expand it into smaller subproblems $P_1, P_2, \ldots, P_k$
  For each $P_i$:
    If $P_i$ is a complete solution: update bestsofar
    else if lowerbound($P_i$) < bestsofar: add $P_i$ to $S$
return bestsofar

- Exact same high level state space search as Partial Path
  - Both typically use priority Queue
  - But they employ different expansion routines
  - Rather than expand each possible next city path (partial path), expand into two new children $P_i$ by including/excluding the edge which maximizes $b(S_e) - b(S_i)$
Search Space for Include/Exclude

$BSSF = 31$ initially

Note that no particular node was constrained to be the beginning node of a path.

After updating $BSSF$ to 21, all remaining states in the priority queue will be pruned.
Include Exclude B&B Complexity

- Number of node expansions required? $O(b^n)$ where $b$ is the average state branching factor ($\leq 2$ and will be less than partial path) and $n$ is the depth of solutions
  - Depth can exceed city number but on average reasonable
- Space Complexity? $O(n^2b^n)$ since we store one reduced cost matrix ($n^2$) for each node in the queue and the queue is the frontier of the search tree which is $O(b^n)$
- Time Complexity?
Number of node expansions required? $O(b^n)$ where $b$ is the average state branching factor ($\leq 2$ and will be less than partial path) and $n$ is the depth of solutions
  - Depth can exceed city number but on average reasonable

Space Complexity? $O(n^2 b^n)$ since we store one reduced cost matrix ($n^2$) for each node in the queue and the queue is the frontier of the search tree which is $O(b^n)$

Time Complexity? – $O(n^3 b^n)$, Since each node expansion tests $n$ possible edges and for each alternative does a cost matrix reduction which is $O(n^2)$
  - The hope is that the extra computation time is made up for by less overall states needing to be considered
Branch and Bound Summary

- Complete – If there is a solution then it will be found
  - As long as there are not infinitely many states with bounds less than the optimal solution
- Optimal – As long as the bound is optimistic it will return the optimal solution
- Upper bound – We have used $BSSF$. In general it could be better to calculate a heuristic upper bound on the solution from any partial state, without having to have actually arrived at a solution. Then any other states with a lower bound greater than the best upper bound anywhere in the space can be pruned.
Implement a time bound
- One advantage of Branch and Bound is you can always just return the BSSF if time runs out – Hopefully it's not the initial one

Use slide example for debugging if you want

Different City distributions
- Easy: symmetric city distances
- Normal: non-symmetric city distances
- Hard: non-symmetric city distances and some infinite distances
  - Use will use the hard setting for all of your reporting

You will report total numbers of states created, max stored, and states pruned

Start early – This program entails somewhat more than the previous ones
State Space vs Search Strategy

- Partial Path has a wider State Space (i.e. search tree)
- That together with the search strategy of taking the highest state from the queue leads to a wide search (more breadth first than depth first)
  - The search tree is static, it is then just a question of our search strategy which decides the order of node expansion
  - A different search strategy (rather than just taking the best score on the queue) could lead to a deeper search to find some early solutions
    - What might we do? – Still can be guaranteed optimal
State Space vs Search Strategy

- Partial Path has a wider State Space (i.e. search tree)
  - That together with the search strategy of taking the highest state from the queue leads to a wide search (more breadth first than depth first)
    - The search tree is static, it is then just a question of our search strategy which decides the order of node expansion
    - A different search strategy (rather than just taking the best score on the queue) could lead to a deeper search to find some early solutions
      - (e.g. PQ key a combination of tree depth and bound)

- Include/Exclude has a binary tree search space, and the nature of include/exclude leads to lower scores down the include side, leading to potential "drilling down"

- Our version still dequeues the best state score and thus drilling will diminish after a while
  - Just like with partial path we could use a different search strategy (e.g. PQ key) to encourage even more "drilling down", thus getting earlier BSSFs with potential for more early pruning.
  - This could lead to "good" solutions that we might be satisfied with once our time limit runs out
In your project, you will note that as the number of cities grow and you hit your time limit, your B&B will almost always return your initial BSSF path.

This is because you are always expanding the node with the lowest score and thus the first actual solution you find, will often be the optimal solution.

However for large $n$ it will take a long time to get the first solution as you continually expand nodes higher in the tree.

There are some extra credit points available if you can demonstrate some effective approaches to encourage some "drilling down" in order to get earlier improved BSSF values.

- Finding a balance, adjust after initial drilling?, round-robin, etc.

Note that this should not sacrifice optimality within the time limit, but when the time limit is exceeded (non-optimal result), you could get a better result than the initial $BSSF$ path.

- You can also gain advantages from early pruning due to tighter $BSSF$s.
Is it Worth All the Work?

- Sometimes it feels like with all the work and overhead that it would be easier just to solve TSP (or other exponential task) with a simpler algorithm.
- Simplest algorithm is to just try each path. There are \((n-1)!\) legal paths for TSP. Each one takes \(n\) adds to compute so total time complexity is \(O(n!)\).
- The Dynamic Programming TSP solution time complexity is deterministic (same time regardless of city distribution) and is \(O(n^22^n)\).
- The Branch and Bound time complexity varies depending on bound, state space, search strategy, city distribution, etc.
- All three approaches give the optimal solution.
- To calibrate, there are about \(10^{57}\) atoms in the solar system and \(10^{80}\) atoms in the current known universe.

### Approximate Time Complexity – big O

<table>
<thead>
<tr>
<th># of Cities</th>
<th>Brute force</th>
<th>(O(n!))</th>
<th>Dynamic Prog</th>
<th>(O(n^22^n))</th>
<th>Branch and Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>(10^6)</td>
<td>(10^5)</td>
<td></td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>(10^{12})</td>
<td>(10^7)</td>
<td></td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>(10^{18})</td>
<td>(10^8)</td>
<td></td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>(10^{64})</td>
<td>(10^{18})</td>
<td></td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>(10^{159})</td>
<td>(10^{34})</td>
<td></td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>(10^{2567})</td>
<td>(10^{307})</td>
<td></td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
A* - Best First Search

- Has many similar properties to branch and bound
- The value of a state/node is $f(n) = g(n) + h(n)$
  - $g(n) =$ the actual cost from the initial/root node to node $n$
  - $h(n) =$ a heuristic value which is an estimate of the cost from node $n$ to an optimal solution
    - For A* to be optimal, this heuristic value must be admissible (optimistic), meaning that $h(n) \leq$ the actual cost from $n$ to a solution
    - Note this is similar to the bounding function for branch and bound
    - A* always expands the node on the frontier with the lowest $f(n)$ value. Fixed state-space search strategy unlike B&B

- Expands a frontier which is guaranteed to
  - expand every node which is less than the optimal solution
  - not expand any node greater than the optimal solution
  - First solution node that it expands will be the optimal solution

- Does not keep a separate BSSF, because the front node on the priority queue implicitly bounds what nodes will be expanded
  - B&B guarantees you won't expand a node greater than BSSF
A* Pseudocode

Based on Artificial Intelligence: by Nils Nilsson

1. Create a search graph $G$, consisting solely of the start node, $n_o$. Put $n_o$ on a list called OPEN.
2. Create a list called CLOSED that is initially empty.
3. If OPEN is empty, exit with failure.
4. Select the first node on OPEN, remove it from OPEN, and put it on CLOSED. Called this node $n$.
5. If $n$ is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from $n$ to $n_o$ in $G$. (The pointers define a search tree and are established in Step 7.)
6. Expand node $n$, generating the set $M$, of its successors that are not already ancestors of $n$ in $G$.
7. Install these members of $M$ as successors of $n$ in $G$. Establish a pointer to $n$ from each of those members of $M$ that were not already in $G$ (i.e., not already on either OPEN or CLOSED). Add these members of $M$ to OPEN. For each member, $m$, of $M$ that was already on OPEN or CLOSED, redirect its pointer to $n$ if the best path to $m$ found so far is through $n$. For each member of $M$ already on CLOSED, redirect the pointers of each of its descendants in $G$ so that they point backward along the best paths found so far to these descendants.
8. Reorder the list OPEN in order of increasing $f$ values. (Ties among minimal $f$ values are resolved in favor of the deepest node in the search tree. Note you could alternatively just implement OPEN as a priority queue).
9. Go to Step 3.
Finding Shortest Path on a Road Map

- What would be a good admissible heuristic?
Find Shortest Drive from Arad to Bucharest Romania with step costs in km
A* search example
Find shortest drive from Arad to Bucharest

- Since Arad is the only node in the queue we dequeue and expand it.
- Expanding a node means calculating $f(n) = g(n) + h(n)$ for each of its children and putting the children in the priority queue.
  - There is no BSSF to compare with, but if $f(n)$ were $\infty$ we would not enqueue it.
A* search example
A* search example
A* search example
A* search example
A* search example
Branch and Bound vs. A*

- A* has the same basic properties as B&B
  - Complete
  - Optimal (as long as we use an optimistic $h(n)$ or bound)
  - Time and Space complexity is $O(b^n)$
- A* can be natural when problem is a path, and B&B when node is an arbitrary state, but both are interchangeable
- A* does not keep an upper bound (e.g. $BSSF$), but ensures that no node is expanded that has $f(n) >$ optimal, whereas BB only guarantees to not expand a node with bound$(n) > BSSF$
  - Thus B&B may expand more nodes than A*
  - However, B&B will not put states on the queue whose bound > $BSSF$. A* will (since it has no upper bound), although those states will never be expanded.
Branch and Bound vs. A*

- A* has a fixed state space search strategy
  - BB can use arbitrary state-space search strategies, including explore a bit, be creative and dig down to find non-optimal solutions, or even get lucky and find the optimal early (though wouldn't know it), etc.
  - This can be nice, especially if you are willing to trade off time and be satisfied with a good solution that may not be optimal
  - B&B has a lower and upper bound for where the optimal solution lies, so you could also stop once that bound gets sufficiently narrow
  - When using a priority queue with the bound as the key for partial path and include/exclude, they search just like A* except for the initial BSSF

- Not enough time? - B&B can return a non-optimal value
  - If there is a timeout at which point a best result needs to be immediately returned, then B&B is good because it just returns BSSF (however, could just be the initial solution)
  - With upper bound B&B can return a BSSF and state that it is within $x\%$ of the optimal solution

- Both A* and B&B can be used with a beam or with non-admissible heuristics to get an approximate solution in more reasonable time/space when necessary
Intelligent Search – When to use

- There are many search problems out there
  - Find the best solution, Find a minimum path, Find optimal set of parameters, Does a solution exist, etc.
- If we can bring some heuristic knowledge to bear regarding bounds and search strategies on solutions then intelligent search will be much faster the un-informed search
- Examples:
  - Game strategy
  - Planning systems
  - Very common in finding solutions for complex AI problems
  - Optimization problems: knapsack, integer linear programming, etc.